Enabling Graphical Variability Modeling for a Software System Across Three Layers of Abstraction

Bachelor Thesis

Timo Günther

2015-12-08

Institute of Software Engineering and Automotive Informatics
at
Technische Universität Carolo-Wilhelmina in Braunschweig (Germany)

supervised by:
Matthias Kowal, M.Sc.
Dr.-Ing. Thomas Thüm

Prof. Dr.-Ing. Ina Schaefer

Abstract

Modeling the variability of software systems can decrease software development costs while increasing its quality but it is not a trivial task. One way of introducing variability is via delta modeling. In that variability modeling approach, a core system is transformed using deltas consisting of the three operation types addition, modification and removal. Additionally, delta-oriented models of software systems can be split up into multiple layers of abstraction. In this bachelor thesis, a partly implemented set of graphical editors for one such three-tiered layer system spanning the activity layer, the architectural layer and the behavioral layer is completed. To this end, two graphical editors are implemented: firstly one for the remaining not implemented layer, the architectural layer, with delta-oriented composite structure diagrams and secondly one for the semantic union of all three layers. The graphical editors are plug-ins for the popular integrated development environment Eclipse based on the Graphical Modeling Framework. The development of the editors is covered from analysis to evaluation. During analysis, other approaches for variability modeling are examined. The results are used in the design phase to derive the optimal means of visualizing delta-oriented composite structure diagrams and mapping models. The inner workings of the graphical editors are explained in the implementation part of this thesis, which also contains discussions about various issues encountered. In the final evaluation phase, the functioning and usefulness of the applications is illustrated with an open case study called the Pick and Place Unit.

Zusammenfassung

Das Modellieren der Variabilität von Softwaresystemen kann Kosten in der Software-Entwicklung verringern und gleichzeitig dessen Qualität steigern, jedoch stellt es keine einfache Aufgabe dar. Eine Möglichkeit zur Einführung von Variabilität liegt in der Delta-Modellierung. In diesem Ansatz zur Modellierung von Variabilität wird ein Kernsystem mittels Deltas bestehend aus den drei Operationstypen Addition, Modifikation und Subtraktion transformiert. Zudem können delta-orientierte Modelle von Software-Systemen in mehrere Abstraktionsebenen aufgeteilt werden. In dieser Bachelorarbeit wird eine teilweise implementierte Menge an grafischen Editoren für ein solches dreistufiges Ebenensystem über Aktivitätsebene, Architekturebene und Verhaltensebene vervollständigt. Zu diesem Zweck werden zwei grafische Editoren implementiert: erstens einen für die letzte nicht implementierte Ebene, die Architekturebene, mittels delta-orientierter Kompositstrukturdiagramme und zweitens einen zur semantischen Verknüpfung (Mapping) aller drei Ebenen. Die grafischen Editoren sind ein Zusatzmodul für die weitverbreitete integrierte Entwicklungsumgebung Eclipse basierend auf dem Graphical Modeling Framework. Die Entwicklung der Editoren wird von der Analyse bis hin zur Evaluation verfolgt. Während der Analyse werden weitere Ansätze zur Modellierung von Variabilität untersucht. Die Ergebnisse dessen fließen in die Designphase ein, um eine optimale Möglichkeit zur Visualisierung von delta-orientierten Kompositstrukturdiagrammen und Mapping-Modellen zu finden. Das Innenleben der grafischen Editoren wird in dem Implementationsteil dieser Arbeit erläutert, wobei auch einige der aufgetretenen Probleme erörtert werden. In der letzten Phase, der Evaluation, wird die Funktionsweise und Nützlichkeit der Anwendung an einer offenen Fallstudie namens Pick and Place Unit veranschaulicht.

Contents

List of Figures

1 Introduction

1.1. Context and Motivation

Maximizing software reuse minimizes development time and cost. With this in mind, software can be designed such that parts of the newly developed software can be reused immediately for further development of similar software systems with multiple variants sharing some means of production. The result is a so-called software product line.

To formalize and document this set of related software variants, it is abstracted into a model. A widespread approach for doing so involves a feature model, that is a representation of all variants regarding which features they possess or lack. However, features are merely labels for more concrete models that can be created by modeling a valid core product first and then working out each variant's differences to it depending on the feature configuration. Those differences are called deltas as they are simply transformations achieved by addition, subtraction and modification to the core product [19].

In addition to variability, software developers might be interested in multiple layers of the system in question in order to capture varying degrees of abstraction. Indeed, various layers have already been explored in the context of delta-oriented modeling, in particular the workflow layer using activity diagrams [23], the architectural layer using composite structure diagrams [13] and the behavioral layer using state charts [15]. The investigation of the interplay between these three layers [16] in turn spawned the idea of a graphical editor combining them for more user-friendly modeling. Thus far, there are two graphical editors that are capable of handling one layer each, specifically one for state charts [17] and another for activity diagrams [1]. This thesis also provides a graphical editor for the final missing architectural layer using composite structure diagrams.

Of course, since each layer provides only a single perspective of the system being modeled, the layers need to work together to form a complete model of the system. This is achieved with the help of a mapping [16]. The mapping describes the relationship between the layers, that is which tasks of the activity layer are handled by which components of the architectural layer and the behavior of which components of the architectural layer are described by which behavioral models. That way, the semantic purpose of one element in one layer is defined in respect to another element in another layer. A complete mapping encompassing all elements thus creates a holistic model describing the entire system instead of just parts of it.

1.2. Goal

The goal of this bachelor thesis is to finalize the layer system of the three graphical editors. To accomplish this, editing capabilities consisting of creation and modification firstly of the final missing architectural layer and secondly of the mapping between the three layers need to be added. The implementation of the mapping should enable the user to make the various layers work together by creating and modifying the model of the mapping. The resulting tool chain should then provide

the ability to develop entire software systems by modeling each of the three layers with its respective graphical editor and subsequently combining them with the mapping editor.

1.3. Structure

Unsurprisingly given the goal of this work, this bachelor thesis is structured like the phases of an archetypal software development life cycle. In chronological order, those are analysis, design, implementation and evaluation. However, before any of that, an explanation of the fundamentals is necessary.

2 Background

This chapter serves as an introduction to the theoretical background of this work. If the concepts explained herein are already known and understood, it may safely be skipped.

2.1. Model-Driven Development

Model-driven development refers to the practice of creating abstract representations of a problem, so-called models, to aid the development of a software system [21]. This reduces both understanding the problem and its solution to more manageable chunks. Since such an abstraction is semantically closer to the problem domain than the solution domain, effort can be outsourced from programmers to domain experts. On top of that, a model defines consistent design concepts and terminology [8], thus making software reuse and communication between developers easier.

Software development is in the unique position where a model is more than just plan and documentation. In particular, models can be used to generate software artifacts such as code and configuration files, or, when taken to the extreme, entire software systems.

That being said, model-driven development only makes sense when creating the model is considerably less expensive than development of the software system. This is given in industrial settings, where multiple developers work on complex software systems.

2.2. Composite Structure Diagrams

The Unified Modeling Language (UML) is a valuable asset in model-driven development [5] that can be used to visually represent software systems in a standardized way using various diagram types. These diagram types are either structural or behavioral. The former describe what the software system consists of and the latter describe how it works.

Composite structure diagrams are a type of structural diagrams consisting of components, ports and connectors. A component represents some architectural unit of the system such as a sensor. A port symbolizes a component's interface through which information can be exchanged. A connector connects ports with each other to denote information flowing from one port to the other. Connectors may alternatively connect ports with the environment, meaning that the environment is sending a signal to the port or vice versa depending on the direction of the connector.

Figure 2.1 is a composite structure diagram with 3 components, 7 ports and 4 connectors. The first component both sends and receives information from the second component. The second component additionally receives information from its environment and a subcomponent while offering a port that is currently unused. The internal workings of the components or when exactly the signals are sent is irrelevant for this type of diagram. What matters is that without having to understand how the components work, the information flow is apparent.

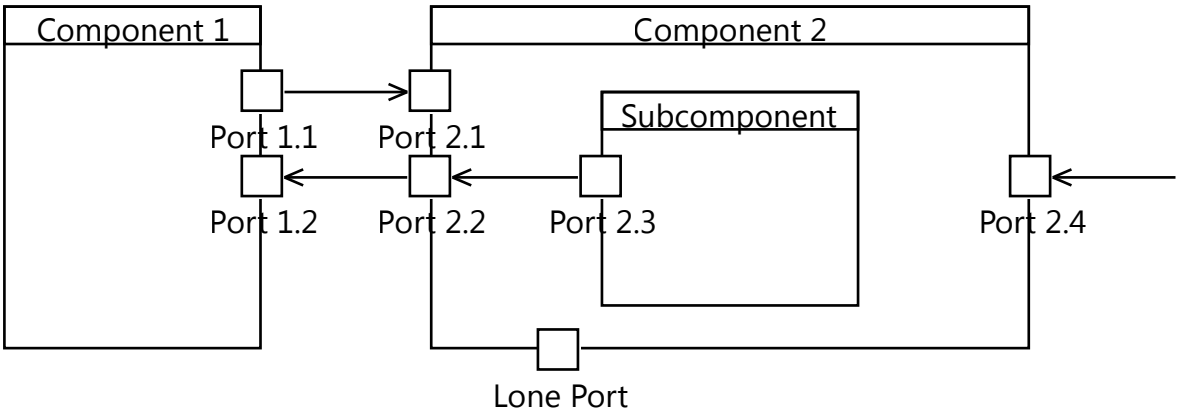

Figure 2.1.: Exemplary composite structure diagram

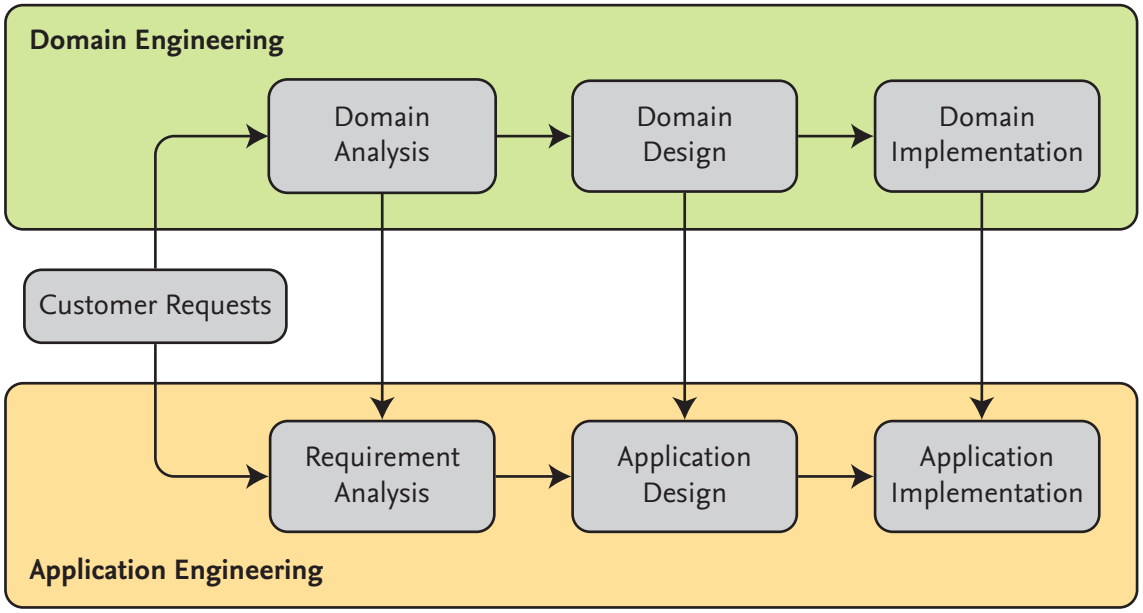

Figure 2.2.: Relationship between domain engineering and application engineering

2.3. Domain Engineering

While in application engineering only a single specific software system is developed, domain engineering is concerned with the multitude of similar software systems which share some means of production and thus profit from planned (as opposed to opportunistic) software reuse [6, 9]. Therefore, to maximize software reuse and minimize the resources spent on application engineering, it is crucial to find similarities and differences between possible software systems of a particular problem domain. The set of these variants is called a software product line.

Figure 2.2 shows the relationship between domain engineering and application engineering. Every phase in classical application engineering also has a corresponding phase in domain engineering that it borrows from. During domain analysis, the domain requirements are determined and the domain model is defined. This should be done with further reuse in mind since the results are used during requirements analysis. The next phase of domain engineering, domain design, takes

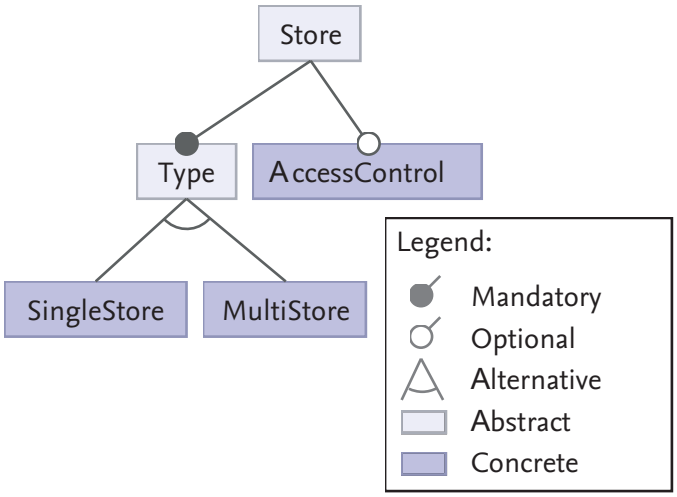

Figure 2.3.: Feature diagram [24]

the domain model defined during domain analysis to create an architecture common to all possible variants of the domain. This architecture should be easily configurable to make extending it during application design as simple as possible. Likewise, in the final step of domain engineering, domain implementation, the general design is implemented so that the resulting artifacts such as program code and documentation yet again eliminate as much work during application engineering as possible.

In an optimal situation, application engineering is as simple as pressing a button to generate the desired variant. However, in practice, it is impossible to find the perfect domain model in one go at the start of the development process. Instead, the domain model undergoes incremental changes by constantly being refined as unforeseen problems or customer requests demand revisions to the domain model.

Due to the abstract nature of the task, implementing a software product line requires thorough planning in form of a model describing the software product line. Modeling a software product line is generally considered to be best achieved using a feature model [11], that is a representation of the software product line saying which feature configurations make up valid variants. A feature model can be expressed in various ways such as text, a propositional formula or an enumeration of all valid feature configurations [24]. The most common representation is a feature diagram like the one in Figure 2.3 as a feature diagram conveys the variability more intuitively and can easily be translated into other forms while this may not hold true for the opposite direction.

2.4. Delta-Oriented Modeling

Delta-oriented modeling is an approach for consolidating domain engineering with model-driven development [19]. Concretely, using delta-oriented modeling, a software product line is expressed as a core model and a number of deltas, which are simply changes to the core model such as addition, removal and modification. Additionally, each delta comes with a condition that must be fulfilled in order to be applicable such as to prevent invalid feature configurations. As a result, any valid variant of the software product line can be created by applying deltas to the core model.

Of course, if there are multiple layers of abstraction, the core model and its deltas need to be

defined on each layer. Additionally, for a complete model, these layers need to be connected semantically using a mapping [16]. In the case of this work, this is done by linking activities from the activity layer with components from the architecture layer and these components again with behavioral models.

3 Analysis

This chapter discusses the necessities of the graphical editors by examining various approaches to model-driven development of software product lines.

Over the years, many such approaches have evolved [2]. They all offer varying degrees of separation between the variability and the base model, that is the instance model or design model. The separation ranges from practically nonexistent to absolute. The following list of modeling approaches starts with the former end of the spectrum and explains them in ascending order of degree of separation.

3.1. Direct Implementation of Software Product Lines

The most basic way to implement software product lines is adding the variability directly to the base model in the solution domain using its own target notation, meaning its domain specific language (DSL). For example, using the DSL for activity diagrams, decision nodes can be used to check whether specific features are enabled in the current feature configuration so that the activity can be changed accordingly. In state machine diagrams, on the other hand, a comparable way of adding variability involves inheritance. Figure 3.1 exemplifies this with a superclass A being extended by various subclasses B, C, D and E providing changes in the form of concurrency, decomposition, additional transitions and additional states respectively. Replacing the superclass with a subclass obviously changes the behavior of the modeled system, so variability can be achieved simply by picking different subclasses depending on the current feature model configuration. Most importantly and in contrast to the other approaches explained below, all of this is modeled using language-level constructs.

The obvious flaw of this approach is that all variability is contained in one monolithic base model. As such, code for variability is scattered and tangled throughout the entire base model. This is a grave concern in the context of domain engineering, as it renders changes to the domain model difficult to realize due to the lack of transparency. Such a complex model also hinders understanding and thus maintaining underlying design patterns. Finally, inheritance hierarchies can grow exponentially.

Speaking of inheritance, the number of DSL-specific implementations of the aforementioned variability mechanism that is inheritance [14, 20, 22] is an example showing that in practice this approach is nonetheless surprisingly common. After all, unlike other approaches, it requires neither an extension of the DSL of the base model (Section 3.2) nor an entirely new layer on top of the base model (Section 3.3). This may make it suitable for but the smallest of projects but does not scale well with the usual complexity of software product lines as it remains too simplistic in the long run.

3.2. Annotations

Variability can be handled in a more sophisticated fashion by making the DSL of the base model capable of expressing more general concepts. Such changes are called annotations due to the form

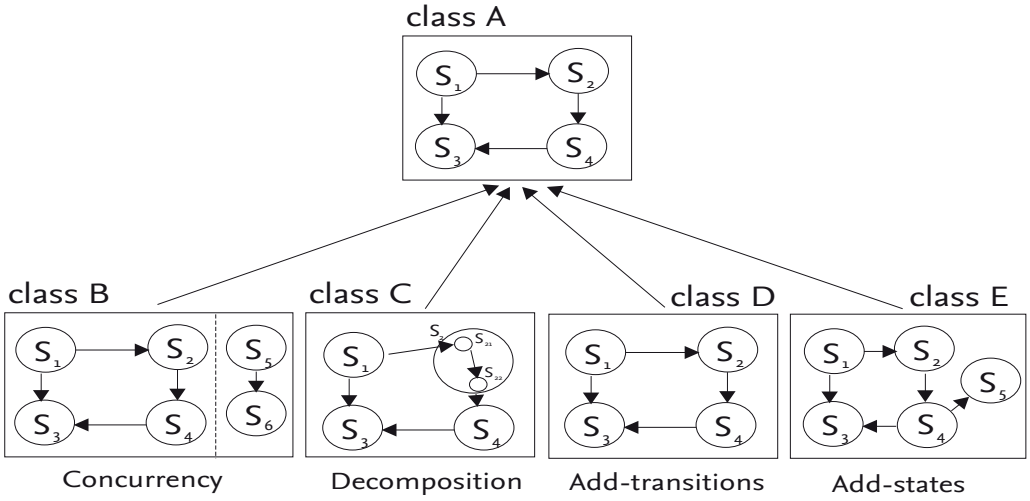

Figure 3.1.: State machine chart extension [14]

they often assume. The resulting model is a union of all variations expressed using the annotated syntax of the DSL. The C preprocessor is an example of this. Using preprocessor constants, part of the code can be included or excluded. At run-time, the same can be achieved using `if` statements.

Annotation-based approaches deal with negative variability. This means that the solution model is different from the base model in that parts not included in the current feature configuration are removed at compile-time, not loaded at load-time or ignored at run-time. In more simple terms, the model starts out big and becomes smaller towards the end of derivation.

The fact that the model is yet again monolithic sheds light on how annotation-based approaches merely alleviate but do not eliminate most of the shortcomings of direct implementation of software product lines. In general, the model becomes hard to comprehend when cluttered with all these annotations [3].

3.2.1. UML-F

UML allows the definition of extensions called profiles by means of changing semantics and syntax to suit the needs of a specific tool or domain. UML-F is a UML profile meant for use in domain engineering and brings a number of constructs that increase the expressiveness in respect to variability [18]. An example of such constructs are the tags `<<framework>>`, `<<application>>` and `<<utility>>` for discerning where classes belong in the software product line semantically. Some other tags are merely representational, like the completeness and incompleteness tags, which denote whether a certain class or some part of it is complete or still needs to be extended to be functional.

Figure 3.2 is an example of how such tagging is represented. The `<<template>>` and `<<hook>>` tags serve to mark methods as template methods or hook methods, the latter of which are internally called by the former. As such, the behavior of a template method can be changed by overriding its hook methods. If a hook method is outsourced to an entirely different class, variability can be achieved simply by supplying classes with differing hook methods to the class with the template method. Thus these two tags can also be applied to classes to denote such behavior. In this example, both the template method `convert` and its hook method `round` are contained in the same class called `CurrencyConverter`. Variable behavior could be achieved by making the hook method round

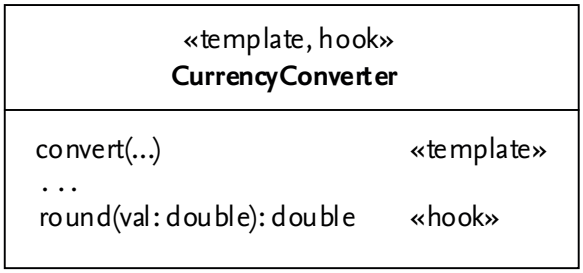

Figure 3.2.: UML-F class diagram [18]

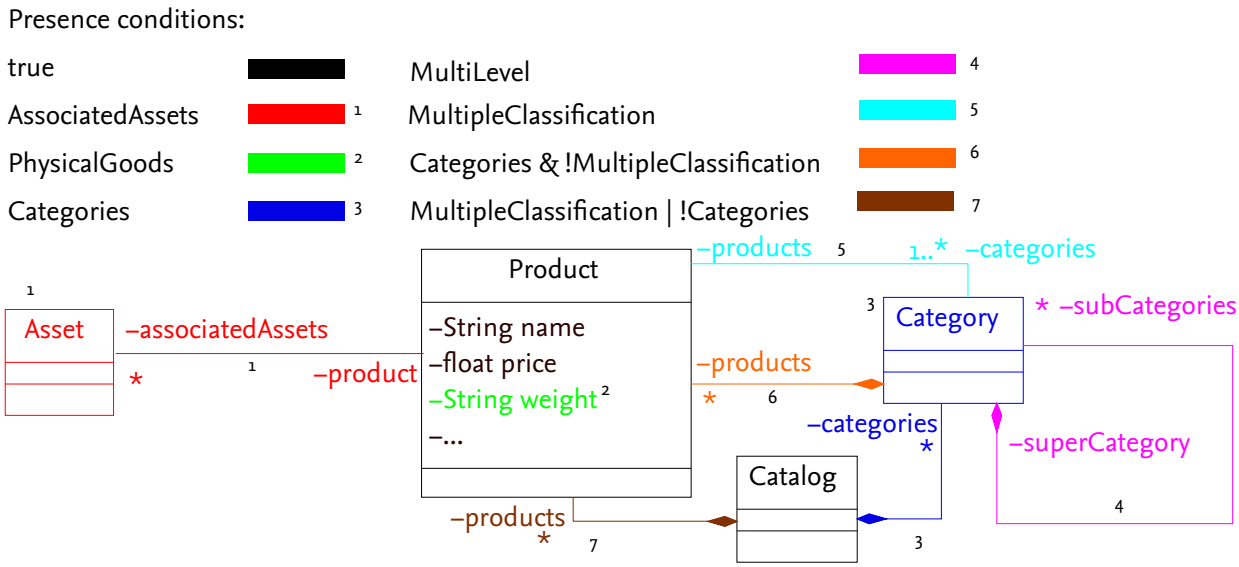

Figure 3.3.: Class diagram annotated with superimposed variants [4]

up instead of down depending on the feature configuration.

Some tag configurations can be replaced by semantically richer ones specifically targeted at modeling such a pattern. The class `CurrencyConverter` could also use the tags `<<Unif-TH: Rounding>>`, `<<Unif-t: Rounding>>` and `<<Unif-h: Rounding>>` to convey that this structure represents a unification of the rounding operation and not just any unspecified template and hook pair. Similarly, entire design patterns can be described with just a few tags.

Such potential for abstraction is one reason to favor UML-F over direct implementation of software product lines. However, closer inspection reveals that the two approaches are not that vastly different in nature. After all, UML-F merely offers the means to sum up the formerly unlabeled patterns a bit more concisely with the use of tags. At the same time, its representation is still lacking, considering that textual annotations all over the model are neither a particularly intuitive nor elegant way to represent variability.

3.2.2. Superimposed Variants

Another approach uses annotations not to model variability concepts directly but to create a mapping between features in the feature model and their manifestation in the base model through various elements [4]. This gives semantic meaning to the features as they are now represented directly in the base model.

Figure 3.3 is an exemplary class diagram annotated with superimposed variants. Each group of elements pertaining to a certain feature is colored in a unique color (and labeled with a unique number to be discernible even if this document was printed in black and white). The presence conditions are derived from the feature configuration and evaluated into simple Boolean values. One might easily be fooled into reading the diagram as if these presence conditions describe which elements get added to the model. Though bearing in mind that annotation-based approaches like this one support negative variability, it is in fact the other way around and they merely describe which ones do not get removed during derivation.

Since every DSL supports the removal of elements for which the corresponding feature is not selected in the current feature configuration, this variant mapping is uniform, meaning it is not restricted to any one DSL in particular, unlike UML-F, which obviously can only be applied to UML diagrams. Another clear advantage of this approach is greatly enhanced feature traceability. It becomes immediately apparent where the variability is.

However, while seeing where the variability stems from is useful, always seeing it is counterproductive. Even in this example with not even a dozen variants, comprehension of the model is hindered by a bombardment of information about all these different variants. Considering that the amount of variants can grow exponentially, having the model show all of them should be avoided. In the case of color coding, there is also the additional issue of potentially running out of easily discernible colors given enough variability.

3.3. Compositions

Composition-based approaches model positive variability and thus add elements where annotation-based approaches would remove them. They take their name from how they decompose a system and its variability by splitting it up into units that may then for example be found in different files. Needless to say, these units must reflect features of the feature model, preferably in a one-to-one relationship. That way, variations are simply composed of a combination of such units. Compared to annotation-based approaches, composition-based approaches are closer to the feature model due to this link between composition and feature configuration.

An important implication is that variability need not be contained in the base model. Rather, the variation model is another layer independent of the base model. Combining the two in a process called resolution finally creates the solution model. Previously, this resolution process was implied with a base model augmented via annotations but is now made explicit.

The resolution of a composition may be tool-based or language-based. The latter requires a common variability language (CVL), which serves the purpose of capturing variability independently of the DSL of the base model [10]. In other words, a CVL is a DSL for the domain of variation models. This is based around the idea that the base model's DSL should stay simple rather than having its expressiveness bent to the point where it encompasses variability. Instead, both the base model and the variation model each have their own unique DSL.

3.4. Transformations

As the name suggests, transformational approaches apply transformations to the model in order to describe variability. Transformational approaches are not limited to annotations or compositions when resolving the model. This provides more flexibility, making them better suited for evolution-

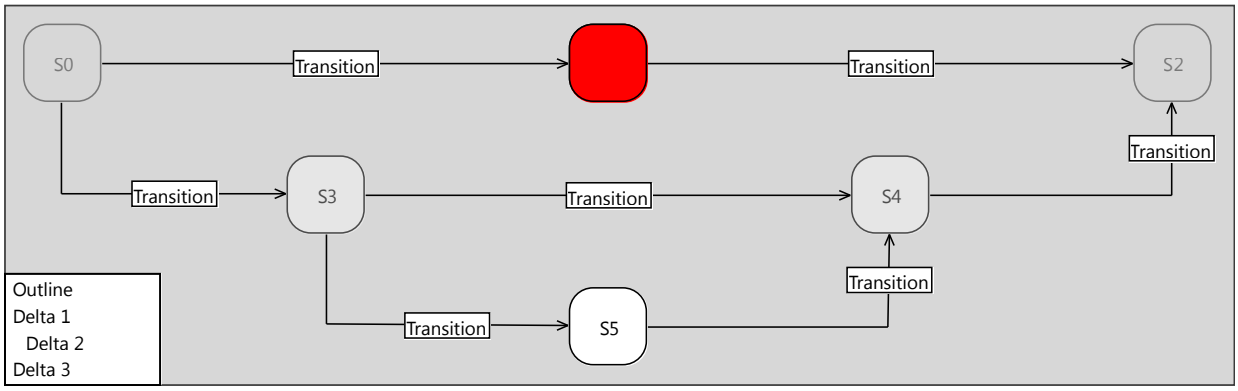

Figure 3.4.: Delta-oriented state chart [17]

ary variability and iterative development. Though with such power comes responsibility. Allowing too many types of transformations may hamper the ease of understanding. It is therefore crucial to avoid bloating the approach with too big a number of transformational constructs.

3.4.1. Delta-Modeling

The idea of delta-oriented modeling is explained in Section 2.4.

An interesting aspect of this approach is that it supports both removal and addition of elements. While the variants are simply compositions of the core model and a number of deltas that are applied to it, the deltas themselves are capable of describing not only positive variability via addition but also negative variability via removal. It is for example possible to emulate approaches for negative variability by starting off with a monolithic core model and only applying deltas that remove elements. Pure positive variability can be modeled analogously. To be fair, surely other approaches can do this more elegantly for only their class, but the point is that delta-oriented modeling is not restrictive in whether to approach variability positively or negatively, so it can take advantage of the best of both worlds. That makes this approach well-suited for iterative development, which requires constant changes both in the form of addition and removal throughout the entire development process [12].

The downside of such flexibility is that it is not immediately apparent whether a given element is part of the application or the variability and thus might still be changed somewhere down the line of deltas. Changing such an element could then cause unforeseen incompatibilities. Such issues could be alleviated given proper tooling support.

Figure 3.4 shows a state chart with its core model and a few deltas. The core model is grayed out to set it apart from its deltas. The changes that its deltas contribute are then highlighted in red in case of removal and in the default color otherwise. This is intuitive as long as the changes from previously applied deltas do not get stacked visibly but are hidden by making it look as if the intermediate model is actually the core model, which this example fails at.

4 Design

Having understood the goal, theoretical background and context of the task, it is now time to plan how to tackle it. This will be based on the previous work on the graphical editors [1, 17]. Much of the technical research and many of the conceptual decisions already made can be applied to this task analogously.

4.1. Programming Language

The new features of the graphical editor should be written in the same programming language that was used for the existing graphical editors, that is Java [1, 17], in order to significantly reduce the effort required to consolidate the various graphical editors later. However, tradition alone is not the only reason to use Java. Many of Java's design goals [7] coincide with the needs for this task. For instance, the graphical editor should not be restricted to any one operating system, so Java's portability comes in handy. On top of that, Java offers a vast array of tools and libraries to employ during development. Particularly noteworthy in the context of this work are Eclipse and its modeling framework.

4.2. Environment

Eclipse is a widespread open-source integrated development environment (IDE) with the philosophy of being easily extendible through the addition of modular features called plug-ins. The graphical editors are such plug-ins [1, 17]. This makes it easy to access the graphical editor in an environment already familiar to many software developers. Moreover, much of the boilerplate code such as creating window frames becomes redundant due to being handled by Eclipse instead.

4.3. Framework

Much effort went into finding the best suited framework for graphical editing of UML diagrams in Eclipse [17]. In the end, the decision was in favor of the Graphical Editing Framework (GEF), which allows iterative development and comes with some useful built-in features such as tool palettes. At the same time, GEF necessitates building the figures by hand, which costs considerable amounts of time but also gives the developer maximum control. Together with the Eclipse Modeling Framework (EMF), it forms the powerful Graphical Modeling Framework (GMF).

4.4. Model-View-Controller

GMF is designed to be used with an architectural pattern called model-view-controller (MVC). As the name suggests, MVC splits a software system into three separate parts respectively called model, view and controller. The underlying principle is the separation of concerns, which helps making software maintainable and reusable. EMF is responsible for the model, Draw2d for the view and GEF for the controller.

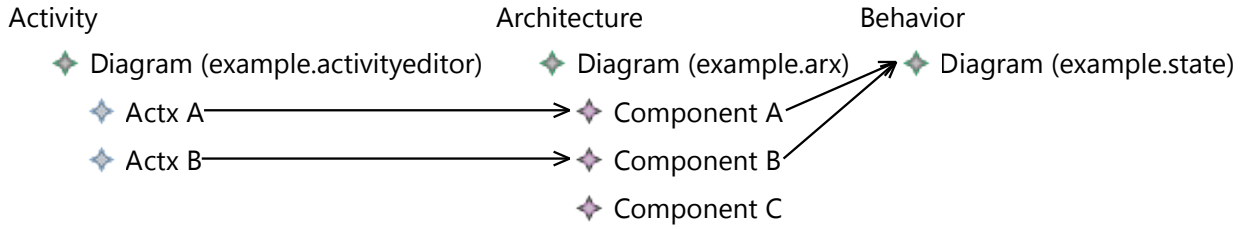

Figure 4.1.: Mapping

4.4.1. Model

The model contains the important data, that is the contents of the model that is currently being modeled using the graphical editor. Concretely, the model of the graphical editor for the architectural layer is a composite structure diagram while the model of the mapping editor is a mapping model.

When using EMF for model-driven development as is the case here, the model is generated by a model and contains another model, so to prevent confusion, these different models need to be differentiated with an explanation of the concepts of concrete syntax and abstract syntax [8]. An abstract syntax can be seen as the blueprint for a concrete syntax. In other words, the concrete syntax is a concrete instance of the abstract syntax; hence the names. Of course, a concrete syntax can also be an abstract syntax when it is used to further create another concrete syntax.

EMF provides the abstract syntax for ECore models. In this work, a concrete syntax of EMF is each used to serve as an abstract syntax for all composite structure diagrams and all mappings, instances of which in turn are meant to be used as an abstract syntax for a software system. In short, a composite structure diagram or a mapping is the instance model, an ECore model is the meta-model and EMF is the meta-meta-model.

Another distinction worth making is the one between the domain model and the diagram model. The domain model contains only the information relevant to the domain; in this case mapping links or components, ports and connectors. The diagram model adds information to the domain model that the view requires; in this case the sizes and locations of the diagram entities. However, some views can function without a persistent diagram model. The tree view for the mapping model in the following Subsection 4.4.2 is an example of this as its canonical layout system allows the sizes and locations of the diagram entities to be computed at run-time.

4.4.2. View

The view is the visual representation of the model. First and foremost, for the graphical editor for the architectural layer, that is a simple composite structure diagram like the one in the aforementioned Figure 2.1. There is not much left to be designed here considering that this is a predefined UML diagram type.

The design of the mapping editor's view offers more freedom. It could be as simple as a text field displaying the mapping links with a more intuitive syntax than how it is stored in the model. Though since the entire purpose of the mapping editor is more user-friendly modeling, it makes sense to go a step further by creating a proper graphical view.

For displaying complex hierarchical data like that from the models of the various layers, tree views

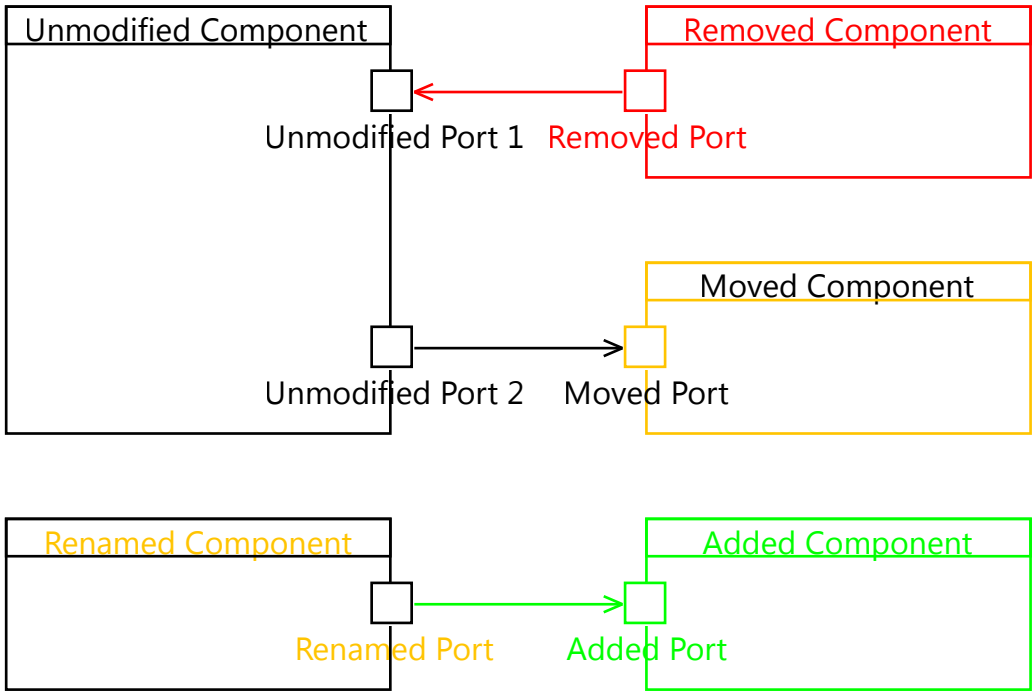

Figure 4.2.: Composite structure diagram with visualized delta operations

are a widespread solution. In interactive media, tree controls also offer intuitive means for shifting the focus of the view by toggling whether a node's children are shown or not. Figure 4.1 shows such a view of a mapping model. The layers are aligned in reading direction from left to right with the tree views of their models spanning orthogonally. The links are connections between tree nodes. The names of the links are omitted to avoid clutter and should be displayed elsewhere, for example in the outline view. Also omitted for the same reason are all elements that are not strictly relevant to the mapping; in other words, only the elements that can be linked or contain other elements that can be linked are shown. In this specific mapping, on the activity layer, two tasks `A` and `B` are each linked to a component with the same name on the architectural layer. These components are then linked to a model of the behavioral layer. Only another component `C` is left semantically useless as it is not linked to anything.

In addition to the simple information displayed in those two views, the delta operations have to be visualized somehow. This should be analogous for both composite structure diagrams and the mapping models. Weighing up the pros and cons of the various variability modeling approaches discovered during the analysis, the most sensible solution is to mark each of the three different types of delta operations in a unique and easily discernible color. With a finite amount of colors, these can be handpicked for maximum recognizability. Figure 4.2 is a composite structure diagram modified by a delta showing the chosen colors. Additions are green, modifications are orange, removals are red and unmodified entities remain in default black. Additions and removals always delete the entity in its entirety, so the whole entity is painted in the respective color. In contrast, modifications only change the color of the affected parts, specifically the name label for name modifications and the geometry for location and size modifications.

When multiple deltas are involved, the view should not be cluttered with the contents of all deltas.

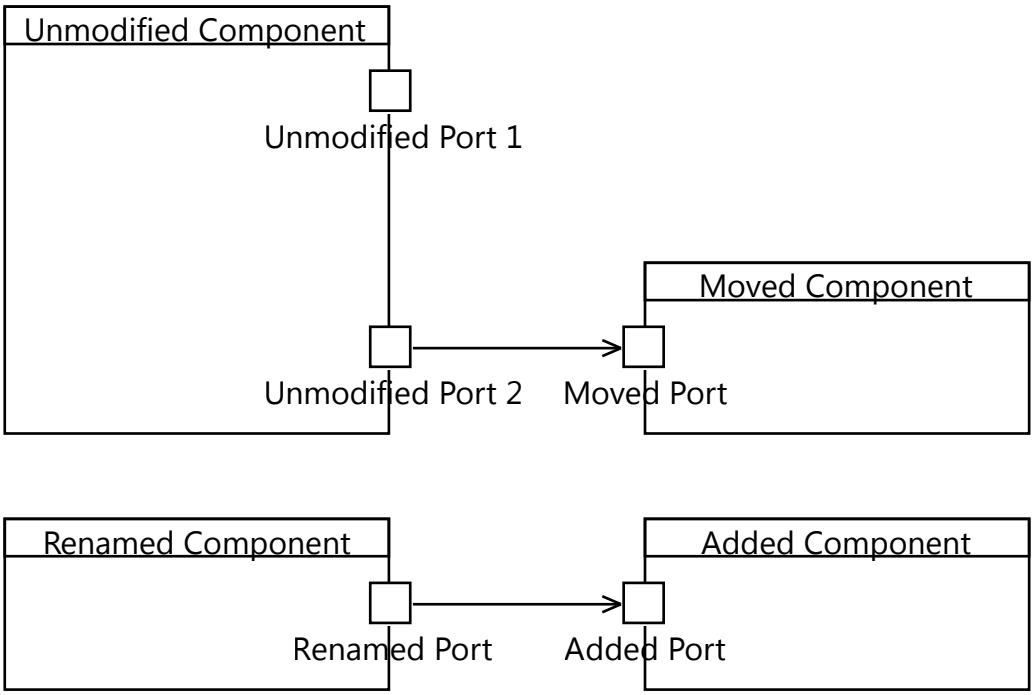

Figure 4.3.: Composite structure diagram with visualized delta operations

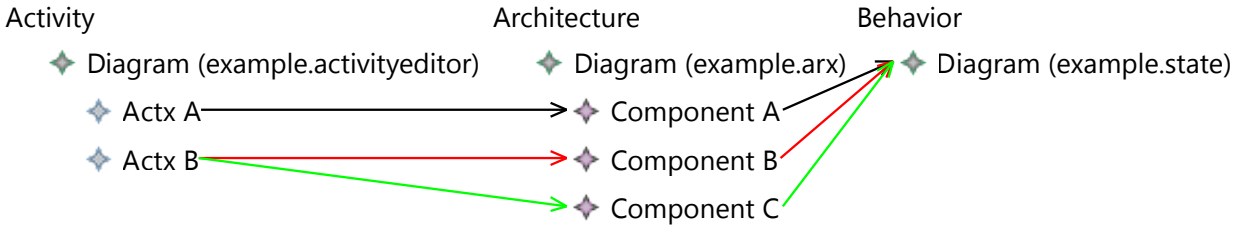

Figure 4.4.: Mapping with visualized delta operations

Indeed, as few deltas as possible should be shown. After all, if a delta is not strictly relevant to another delta, it just distracts from it. Despite there being a few exceptions to the usefulness of this rule like for example during comparison of deltas, it is generally a sound guideline.

Of those deltas that absolutely need to be visible, that is the currently viewed delta and the ones it depends on, only the leaf of the delta hierarchy should be visualized as above while the other ones are resolved to look as if they formed just another core diagram together. Masking the information about other deltas in such a fashion ensures that there is no confusion as to whether some operation belongs to the currently viewed delta or some other one. Figure 4.3 is the composite structure diagram from Figure 4.2 with resolved delta operations. This is what it would look like if another delta depending on the previous delta were to be viewed. This is also what it would look like if multiple deltas were to be resolved together.

An example of a delta-oriented mapping can be found in Figure 4.4. The mapping from Figure 4.1 is changed in such a way that the task `B` is now handled by component `C` instead of `B`.

4.4.3. Controller

The controller handles the interplay between the model and the view. The controller understands the user's request to modify the model and commits it. When the model is changed (by the controller or any other source), it notifies the controller of the change. The controller thus updates the view to reflect the current state of the model. The user, seeing the updated view, may then decide to request further modifications to the model by interfacing with the controller.

5 Implementation

This chapter outlines the implementation of the graphical editors. This should give an idea of why the graphical editors work the way they do and are structured the way they are, so this is done semantically as opposed to syntactically where possible. Included are also descriptions of some of the more specific difficulties that arose during the implementation of the graphical editors to give future developers of the graphical editors an idea of how to approach similar issues.

The implementation is split up into various Eclipse projects. Each one of them provides an Eclipse plug-in as well as a Java package using the root `de.tubs.cs.isf` in accordance with Java package naming conventions. Creating plug-ins for Eclipse is done fairly easily by hooking into the so-called extension points provided by Eclipse. A project's plug-in file (`plugin.xml`) declares such extensions among others. Following is an explanation of the projects. However, since the implementation of the mapping editor turned out to be very similar to the one of the graphical editor for the architectural layer, the explanation of the latter usually suffices to also explain the former and is as such not repeated.

5.1. EMF Model

This project is for the model part of the application handled by EMF as elaborated in Subsection 4.4.1. The model folder (`model`) contains two EMF files from which the model's Java code in the source folder (`src`) as well as plug-in settings are generated.

5.1.1. ECore Model

One of the EMF files is the ECore model defining a number of ECore types capturing the structure of delta-oriented composite structure diagrams (`arx.ecore`) or mappings (`mapping.ecore`).

Simple Diagrams

Intuitively, a simple, non-delta-oriented composite structure diagrams is a collection of diagram entities, that is components, ports and connectors. This is reflected in the ECore model with the ECore types `Diagram`, `Component`, `Port` and `Connector`. The root of the containment hierarchy is `Diagram`, of which there should be exactly `1` instance per ECore model instance. An instance of `Diagram` may contain any amount of instances of `Component` and `Port`. The reason why instances of `Diagram` may contain not only instances of `Component` but also instances of `Port` is explained in subsubsection 5.1.3.

In an earlier version of the ECore model, instances of `Connector` were contained in an instance of `Diagram`. However, that modeling approach requires instances of `Diagram` to store an additional list for contained instances of `Connector` even though instances of `Port` already keep track of incoming and outgoing instances of `Connector`. Instead, one of the latter two lists can simply be turned into a containment reference. That eliminates the need for a redundant list in `Diagram`, preventing issues such as data inconsistencies when left uncontrolled. Indeed, each instance of `Connector` is now contained in its source node, that is an instance of `Port`. The choice between source node and target

node here is arbitrary.

Just as with `Diagram`, an instance of `Component` may also contain any amount of instances of `Component` and `Port`. As this is the same containment hierarchy as for `Diagram`, it is outsourced to the ECore type `Container`. On top of that, `Component` and `Port` are unified as extensions of `Containable`; instances of which, as the name suggests, are contained in an instance of `Container`.

This does not yet make the ECore model complete, as components and ports, that is instances of `Containable`, still need to be named and positioned. This is handled with the ECore types `Nameable` and `Positionable`. The former simply stores a string. The latter adds the integers `x`, `y`, `width` and `height`. These integers could instead be stored more concisely in a single instance of `Rectangle`, but teaching EMF how to serialize that into string format would require manipulating the generated code. Using primitives and adding utility operations for conversion between the types is the more elegant solution. The addition of these utility operations can be achieved by supplying the ECore model with either the operations together with annotations containing the method bodies or with a transient and volatile attribute of the type `Rectangle`. The latter option, while no doubt conveying more meaning through the model, is ruled out yet again due to requiring changes to the generated code.

The EMF model for simple mapping diagrams is similarly structured. The main difference is that links are the only diagram entity type, so `Diagram` and `Link` are already all of the concrete types. Links are like connectors except they do not connect ports but any elements from other referenced models. Since therefore its nodes have to be of the generic object type, instances of `Link` cannot be contained in either of its nodes but in an instance of `Diagram`. This is achieved simply by making `Link` a subclass of `Containable`.

Delta-Oriented Diagrams

Building upon simple diagrams are the delta-oriented ones. The initial idea for modeling delta-oriented composite structure diagrams or mappings is comparable to typical object-oriented extension, where the diagram entities are the features that can be extended. The core diagram is the superclass and its deltas the subclasses, which of course can be extended as well with yet more deltas. An empty delta making no changes would have the same contents as the core diagram. An addition is modeled by a delta containing an entity not already contained in its core diagram. Analogously, a removal is modeled by a delta not containing an entity that is contained in its core diagram. Modifications, on the other hand, are achieved by having the value of an entity's feature differ between the delta and its core diagram.

An issue presents itself here when the core diagram changes. Simply copying the core diagram into the delta leads to redundancy with all of its common issues. The copy in the delta needs to be updated when the core diagram changes, else the delta suddenly contains an unintended modification. So instead of copying all of the diagram entities, a delta's entities should be empty and only update the corresponding core diagram's entities where necessary. This is the same as in object-oriented programming, where a subclass may not specify an overriding method body, in which case the method body of the superclass is used.

The question then becomes how to tell whether an entity's feature is empty and undefined or overridden to a new value. A programmer's first thought when hearing of undefined values is of course the value `null`. That would make it impossible for a delta to set a diagram entity's feature to

`null`, though, should that value possess meaning for the feature in question because, as far as the delta is concerned, the value was never overridden. Luckily, EMF comes with the ability to declare features as unset. This is similar to but not the same as `null`, though on closer inspection this is just moving the goalposts as a delta may wish to commit the modification that a entity's feature shall be unset. As such, the only proper solution is keeping a map of all features that are actually modified even if the value happens to be the one that denotes an undefined value.

That makes this delta-based extension mechanism accurate for modifications, but not for removals. While the redundancy stemming from copying the values of all features of each diagram entity is eliminated, the redundancy caused by copying the references to the entities in the delta remains. Upon addition of an entity to the core diagram, the delta therefore adds an unintended removal considering that this reference is missing from the delta, which is the condition for something to be considered removed in a delta.

This could be counteracted by updating each delta when entities are added to or removed from the core diagram. However, by now it should be apparent that controlling all this redundancy is quite the hassle. Moreover, the described delta-based extension mechanism has yet to support multiple extension in case a delta manipulates diagram entities from multiple core diagrams. There must be a better way to model deltas.

Indeed, the solution is to look at deltas and model them more closely to their formal definition [19]. A delta really only is a list of operations, that is additions, modifications and removals. Sticking to that, the ECore types `Delta`, `Operation`, `Addition`, `Modification` and `Removal` are introduced. In addition to the previous containment references, an instance of `Diagram` may now also contain any number of instances of `Delta`.

An instance of `Delta` should contain any number of instances of `Operation`. Since EMF does not support feature refinement, as explained in subsubsection 5.1.3, but specialization is necessary, an instance of `Delta` in truth may contain any number of instances of the concrete subclasses of `Operation`, namely `Addition`, `Modification` and `Removal`. An attempt was made to solve this with EMF feature maps[1], though it remains unclear how a feature map can function as a containment reference considering that it is not a reference but an attribute.

An instance of `Removal` merely holds a reference to the target it deletes. To this end, everything that can be removed by a delta is denoted by the ECore type `Entity`. For mappings, the removable types are only `Link`, and for composite structure diagrams, those are `Component`, `Port` and `Connector`, so it is a union type of `Containable` and `Connector`.

An instance of `Modification` also holds a reference to the target it modifies. Moreover, it stores the key to what attribute it modifies, which is one of the constants from the generated `ArxPackage` or `MappingPackage` classes, as well as the new value of the attribute.

An addition yet again holds a reference to the target that it adds to and contains a payload that it adds to the target. Since the payload may be any diagram entity, `Addition` extends `Container` and, in the case of composite structure diagrams, the newly introduced type `Node` that outsources the ability of `Port` to contain instances of `Connector`. Type-wise, the target may be anything, since unifying `Container` and `Port` does not create a sensible type hierarchy. Here once more feature refinement would come in handy in conjunction with making `Addition` abstract and creating concrete subclasses for the different types of targets and payloads. Same applies to the cardinality of the containment

[1] *EMF FeatureMaps*, 2004-06-24: `http://www.eclipse.org/modeling/emf/docs/overviews/FeatureMap.pdf`

references of `Container` and `Node`; as of now, an instance of `Addition` may unfortunately contain multiple payloads.

Even though this delta-modeling solution is hindered by the lack of feature refinement in EMF, it is nonetheless much cleaner than the previous one. Deltas are modeled without butchering the simple diagram model with maps, additional references or changes to the range of values but with just a few more types.

5.1.2. GenModel

The other EMF file in the model folder is the GenModel (`arx.genmodel` or `mapping.genmodel`). It is linked directly to the ECore model just described in Subsection 5.1.1 and is used to generate the code based on the ECore model. It can be configured with a number of settings such as using generic Java lists instead of internal EMF lists.

Throughout the entire implementation, care was taken not to rely on changes to the generated code for the application to work. This means all the files that are generated by EMF, including this project's code (but not the two EMF files in the model folder), can safely be deleted and replaced with newly generated versions.

5.1.3. Difficulties

The following describes modeling demands that EMF does not meet and issues to which the solution might not be entirely obvious right away.

Feature Refinement

A constraint of EMF is that feature refinement is not supported[2], meaning specializing an attribute of a superclass in a subclass is not possible[3]. From a technical point of view, EMF treats features of the same name and type as distinct features and will complain about duplicate attribute names instead of overriding the properties. Java allows neither overriding the cardinality nor the generic type contained in a collection, so this makes sense for the cardinality and type properties but less so for the other constraints modeled with EMF such as transience and uniqueness.

One way feature refinement would be useful in the graphical editor is the abstraction of diagram entities with a size and location. All of these extend the ECore class `Positionable`. With feature refinement, some subclasses could change properties such as the default size or whether the size can be changed at all. Indeed, components should by default be greater in size than ports and ports should never differ in size.

This can be done by adapting the Java code generated from the ECore model, but this is avoided to keep the model in alignment with the code it generates. Another option is moving all relevant features from the root of the inheritance hierarchy to its leafs and directly resolving the refinement, which of course also removes the possibility of abstracting properly because the features are missing from the superclass. This presents the dilemma of having to choose between abstraction and specialization. The graphical editor tends to take the approach of abstraction in hopes that the amount of specialization is negligible for this application.

[2] *Is Feature Refinement Supported?*, 2006-10-24: `https://www.eclipse.org/forums/index.php?t=msg&th=130652`
[3] *ECore "Abstract Attribute"*, 2012-05-26: `https://www.eclipse.org/forums/index.php?t=msg&th=355799`

Connections Without Endpoints

In composite structure diagrams, a connector is a connection between ports or, and this is the catch, between a port and the environment. The environment makes for a particularly unfit endpoint for a connection as it is not even a diagram entity.

In an intuitive solution approach, the environment could be represented by the root of the component that contains the port, that is the diagram. This complicates creating and changing connectors, though, since it makes the diagram responsible for remembering the start and end locations of the connectors.

Instead, this information can be outsourced to a new diagram entity. This is done with the introduction of signals, which can be seen as environmental ports that need to be modeled explicitly. This means connectors now connect ports with ports or signals.

Additionally, ports and signals can be merged into the same diagram entity by changing the semantics of ports such that they may also be added to the environment (meaning an instance of `Diagram`) and not just to components. To this purpose, the ECore types `Container` and `Containable` are introduced. The former may contain any amount of the latter. This abstraction is convenient as it also makes containing components inside components as simple as making `Component` extend both `Container` and `Containable`.

5.2. Edit

This project is generated from the EMF files explained in Section 5.1. It provides classes useful for presenting the model in a number of different formats such as a tree. The other projects make use of this.

5.3. Generated Editor

This project is generated from the EMF files explained in Section 5.1. It provides a generated editor for the model, which is obsolete with a full-fledged graphical editor already implemented, but might still come in handy for debugging. It also adds the wizard for creating a new delta-oriented composite structure diagram, which has not been recreated as that consists mainly of boilerplate code for opening files and wizard pages.

5.4. Graphical Editor

Since each of the two graphical editors is an editor, it declares an extension for the editor extension point containing an editor definition consisting of various attributes, among which is the main editor class to be instantiated when the plug-in is started: `ArxEditor` or `MappingEditor`. Being an extension of the GEF class `GraphicalEditorWithFlyoutPalette`, which more importantly is in turn an extension of `GraphicalEditor`, it comes with some methods for integration into the Eclipse environment. Given a resource, usually a file, to work with, `ArxEditor` or `MappingEditor` is responsible for loading and saving the editor's contents, which is the model object at the root of the containment hierarchy.

GEF automatically creates an instance of `EditPart` upon loading the contents. GEF knows which type of `EditPart` it needs to instantiate because `ArxEditor` or `MappingEditor` sets its own instance of `EditPartFactory`, specifically `ArxEditPartFactory` or `MappingEditPartFactory`. How this puts the rest of

the editor in motion is explained in subsubsection 5.4.3.

`ArxEditor` or `MappingEditor` is also where properties like grid spacing (which for `ArxEditor` by default is the default size of instances of `Port`) are set. When asked for an instance of `IContentOutlinePage`, it returns an instance of the inner class `ArxContentOutlinePage` or `MappingContentOutlinePage`, which adds the undo, redo and delete actions to the context menu. Similarly, `ArxActionBarContributor` or `MappingActionBarContributor` globally enables these actions within Eclipse for when the editor is active.

As mentioned already, `ArxEditor` or `MappingEditor` extends `GraphicalEditorWithFlyoutPalette`, a class provided by GEF that conveniently adds a tool palette to the editor without additional effort. Using `ArxPalette` or `MappingPalette`, this palette is then modified to contain the tools necessary for working with delta-oriented composite structure diagrams or delta-oriented mapping models. In particular, these are a selection tool, a creation tool for each of the entity types and a creation tool for deltas.

When the selection tool is selected, figures shown in the editor can be selected using the mouse. GEF handles figuring out which figure was selected and which instance of `EditPart` that figure belongs to. When actions such as selecting, resizing or removing are taken, an instance of `Request` is sent to the various instances of `EditPolicy` registered to that instance of `EditPart`. Depending on the type and contents of the request, this determines the behavior of the editor.

Each creation tool relies on an instance of `CreationFactory` to create a model instance of the respective type when selected. This model instance is packaged in an instance of `CreateRequest` which then undergoes the same ordeal as any other instance of `Request` from the selection tool.

`ArxEditor` or `MappingEditor` is also the class that stores which delta is being edited by keeping track of selected deltas in an instance of `DeltaConfiguration`. Which deltas are also activated along with the selected deltas is determined by whether or not other deltas depend on the selected deltas. A delta depends on another delta if any one of its operations targets an addition from that delta as otherwise it would be impossible to resolve the operation because the target is not visible in the diagram.

5.4.1. Model

Even though EMF is responsible for almost everything related to the model, a small amount of work has to be put into making GEF capable of manipulating the model.

Command

Instances of `Command` are created by instances of `EditPolicy` in response to receiving instances of `Request`. Each `Command` may be executed to invoke changes in the model. Typically, this involves setting some attribute or reference through the code generated by EMF.

Early on during development, instances of `Command` would do more than such minor changes. For example, an instance of `DeleteContainableCommand` would not only delete one instance of `Containable` but also delete possibly existing child entities of that instance. However, this is work best left to the `EditPolicy` by creating additional instances of `Command`. After all, the model should not know anything about the controller, yet information from the controller is sometimes required to properly manipulate the model. For instance, only the controller has access to the critical information which deltas are currently active in the editor.

CreationFactory

The sole purpose of a `CreationFactory` is to create model instances of a certain type. With EMF, all it takes to instantiate a type is to delegate to the generated factory class `ArxFactory` or `MappingFactory`. Obviously no more and no less than all the concrete model types can be instantiated, leaving only `ComponentCreationFactory`, `PortCreationFactory`, `ConnectorCreationFactory` and `DeltaCreationFactory` for composite structure diagrams and `LinkCreationFactory` for mapping models. However, a subclass called `DiagramCreationFactory` does not exist as creating a new diagram should involve the creation wizard instead of a creation tool.

5.4.2. View

For every model instance exists a representative Draw2d figure. The view consists entirely of specialized Draw2d figures that in turn consist of more basic figures such as rectangles. Each figure is kept up to date by a corresponding instance of `EditPart` that always sets the figure's attributes to be representative of the model instance and then repaints the figure.

The figure interfaces are separated from their implementation to enable reuse. The implementations of the figures can easily be swapped to change the appearance of the editor's contents.

In general, implementation details of the view should not seep into the controller. For example, when updating the figure, the controller could simply change the color of the figure depending on its delta state using `setForegroundColor(Color)`. However, to keep the code modular, the controller should not know anything about how the delta state is represented in a figure. Therefore, the controller should instead merely let the figure know what its delta state is so the figure can choose its own appearance without outside influence.

5.4.3. Controller

The controller is a complex mechanism held together by GEF. Its general function is explained in Subsection 4.4.3.

EditPart

Edit parts are the basic building blocks of GMF and the glue holding the model and the view together. The very first edit part to be instantiated is the `RootEditPart`, which is the only edit part that does not have a corresponding model object. This special edit part exists so that every normal edit part has an instance of `EditPart` as parent, normalizing the behavior. The only child of the `RootEditPart` is called the contents, which is loaded together with the editor.

To create edit parts for model objects such as the contents, the editor relies on an instance of `EditPartFactory`, namely `ArxEditPartFactory` or `MappingEditPartFactory`, which checks the type of the model object and returns a newly created edit part of the correct type for the calling edit part to set as child.

After being added to the `RootEditPart`, the edit part for the contents does the same as what the `RootEditPart` just did: creating an edit part for each model child. Thus the editor recursively populates itself with edit parts, each one of which is tailored to some specific model object.

The view's figures come together with the controller's edit parts. Each edit part knows which type of figure it needs to instantiate and how to update it when model changes occur.

An edit part may declare to have different model children than it actually does from the perspective of the model containment hierarchy. This does not cause issues within GEF; on the contrary, it is

handy for loading data not contained by that model object. For example, the children of `Diagram` of the mapping editor will always artificially include the values of the ECore enumeration `MappingType`, specifically `actx`, `arx` and `statecharx`, in order to create edit parts specifically designed at creating tree controls for models of the respective layer of abstraction.

The same mechanism is used in handling deltas. The resolving of deltas works through having the edit parts adding and removing model children from its list of model children depending on whether or not that model child is added or removed via one of the active deltas. To be precise, though, the model children of entities removed in just the edited delta but not in any of the deltas it depends on are not removed from the list of model children. Otherwise, their figures would simply vanish from the diagram instead of staying there to be marked in red later.

The resolving of modifications works similarly. When the figure is updated, instead of directly using the value the model object has stored, the active deltas are checked for modifications on that model object's feature. If a match is found, the value in the modification is used, making it look like that is the actual value of the model.

The concrete subclasses of `OperationEditPart` only exist to refresh the editor when their models are modified. One might think that it therefore suffices to make `OperationEditPart` an `EditPart` instead of a `GraphicalEditPart` to save oneself the trouble of defining empty figures. Regardless, GEF will crash without corresponding figures because it assumes that every child of an `AbstractGraphicalEditPart` is also one.

As a final remark on edit parts, due to the close link to the model, the edit part hierarchy mirrors that of the model; however, while EMF supports multiple inheritance, Java does not. For this reason, some of the types in the EMF model can only be recreated as interfaces but not as classes. This causes some code duplication.

`Request`

Instances of `Request` are sent to instances of `EditPart` and `EditPolicy` to start controller processes such as creating and executing specific instances of `Command`. For example, `ContentsLayoutEditPolicy` uses `ChangeChildBoundsRequest` to differentiate between the original `ChangeBoundsRequest` and the ones it created itself to move the children together with the parent.

`EditPolicy`

Edit parts can assign instances of `EditPolicy` to certain tasks such as managing the layout or creating deletion commands. This is pretty straightforward for simple diagrams but gets complicated when deltas are involved.

The common pattern is the following. First is checked whether multiple deltas that are leafs in the delta dependency tree are active. In this case, all editing is disabled to keep the editor in a consistent state while the resolved view is shown. If currently no delta is active, it suffices to employ the usual approach for simple diagrams involving direct manipulation of the model. If on the other hand a single delta is active, the delta approach is required. What exactly the delta approach entails depends on the task at hand.

When creating a new figure, the delta approach means creating and adding a new addition to which the figure is added. Care must be taken not to stack multiple entities in the same addition. If the parent is removed in an active delta, adding to it is disallowed.

When removing existing figures, the delta approach means creating and adding a new removal

targeting that figure. However, if the target is removed in any active delta, no removals are created; on the contrary, if applicable, the existing removal of the active delta is undone by deleting it. If the target is added in the active delta, the addition is undone by deleting it. All of this needs to be repeated recursively for the children figures as well.

When modifying an existing figure, the delta approach means creating and adding a new modification targeting that figure. However, if the target is added in the active delta, no modifications are created; instead, the existing addition is modified directly. If the target is removed in an active delta, modifying it is disallowed. For repositioning but not for renaming, all of this needs to be repeated recursively for the children figures as well.

`EditManager`

An `EditManager` fulfills a similar function like an `EditPolicy`. The only `EditManager` used in this application is `NameableDirectEditManager`, which allows renaming of instances of `Nameable`, that is `Component` and `Port` for composite structure diagrams and `Link` for mapping models, directly inside the diagram. To this end, it uses the inner class `NameableCellEditorLocator` to position the editing field. Once the editing field is positioned and shown, a new name may be entered for `NameableDirectEditPolicy` to use in a change name command.

5.4.4. Difficulties

During the implementation of the graphical editors, a handful of considerable difficulties arose. The more specific ones are listed here so that future developers of the graphical editor have the corresponding solutions already at hand.

Z-Order of Child Figures

Draw2d handles the order in which figures are painted. Naturally, one would expect that GEF would add the figures in such a way that the z-order of a child figure is always higher than the one of its parent to convey the sense that the child figure is inside or on top of the parent. However, GEF paints depth-first [8], meaning that the child is painted before the parent is. This results in the unfavorable situation of a parent covering its child figures, making it look as if the child figures were not painted at all.

The symptom of invisible child figures can be eliminated by preventing the parent from painting over areas that are critical to the children. The easiest way to accomplish this is by making the parent's background transparent by calling `parent.setFill(false)`. However, this is not necessarily intended behavior. After all, with no background fill, figures become hard to see when their foreground color is similar to the color beneath them. Exporting the diagram to be displayed on a website with a black background for example would thus make the diagram invisible.

A solution is to not use child figures at all. Instead, all child figures get added directly to the contents of the window instead of some of its children. That way, the order in which figures are painted can be influenced directly. Of course, this wastes the features for child figures that GEF brings, so operations such as movement of child figures have to be recreated by hand. It also creates a semantic gap between GEF figures and the model they represent.

Fortunately, the z-order can be changed by changing the order of child visuals in the edit part. The method `EditPart#addChildVisual(EditPart, int)` fetches the figure for the given child and adds it at the given index. Overriding this to always add to the end of the list of figures means early figures

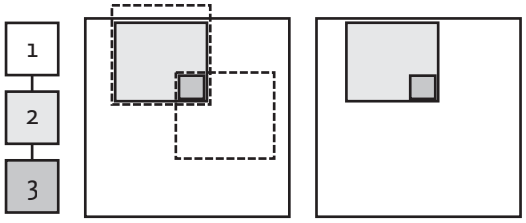

Figure 5.1.: Clipping of figures in Draw2d [8]

like the parent figure are added and therefore painted before more recently added figures like the child figures.

Child Figures Extending Over Parent's Constraint

For each figure in the scene, Draw2d stores a constraint denoting which area of the scene that figure takes up. Knowing which areas have to be repainted is critical to the performance of the painting algorithm. Of course, this only works as long as figures really only are as big as they announce to be. As such, a figure attempting to paint outside of its constraint will be truncated. The same clipping is also applied to all of its child figures [8]. The resulting behavior is visualized in Figure 5.1.

Needless to say, such clipping is problematic when having child figures extend over the parent's constraint is intended behavior. In composite structure diagrams, this is exactly the case with ports, which are visually both inside and outside of the parent component. Due to the way Draw2d clips figures, only the part of the port that is inside the parent is actually painted.

This problem can be avoided with the same approach as the sloppy solution to the previous issue. Adding all child figures to the contents of the window means that they all get to paint in their own constraints regardless of what the constraint of the figure for the model's parent are.

Alternatively, the parent's constraint can be increased in size to accommodate for the child figures. While this works, it of course means that the parent's constraint has to be computed manually, which requires knowledge of the constraints of the child figures. On top of that, the child figures too need knowledge of the parent figure in order to counter the change in the local coordinate system. Such reduced modularity is unfortunate. This seems more like a job for the figure's layout manager.

Indeed, there is a layout manager specifically targeted at solving this problem. `FreeformLayout` is meant to be used in conjunction with instances of `FreeformFigure`, which are figures that expand depending on the size of their children. As that turns the entire working order of the layout algorithm upside down, its usage is unusual. For instance, the constraint of the figure cannot be defined directly as it has to be calculated from the children constraints. While overcoming these peculiarities can prove difficult, the rewards are well worth the effort.

Tree Controls Inside `GraphicalEditor`

The mapping editor relies on tree controls inside its main editing area. Typically, creating a tree control using GEF involves instances of `TreeViewer`, which are used in conjunction with `TreeEditor`. However, while there is a class `AbstractGraphicalEditorWithFlyoutPalette`, there is no analogous class `AbstractTreeEditorWithFlyoutPalette`. Additionally, the mapping editor requires multiple tree controls for the three layers while `TreeEditor` only contains a single tree control. An option is to override the implementation to split up the editing area into several smaller ones that each contain their own tree control. However, then these areas are disjoint, making it impossible to for example

have a line start in one area and end in another, which is precisely what the links of the mapping editor need to do. Therefore, the mapping editor cannot use `TreeEditor` but needs `GraphicalEditor`.

So perhaps it is possible to use `TreeViewer` directly within `GraphicalEditor`. Unfortunately, the two are incompatible. Not even within the same underlying `TreeViewer` can a connection figure be painted because, unlike instances of `GraphicalEditPart`, instances of `TreeEditPart` do not even use figures to get painted. This means that the tree controls have to be rebuilt from scratch inside `GraphicalEditor`.

The bare minimum for a functional tree control is a hierarchically structured view with the ability to expand or collapse nodes. The view of a tree control is done fairly easily by aligning the nodes vertically and giving each figure a horizontal indentation. Since the child figures are positioned relatively to the the parent, the indentations stack, resulting in the typical tree look. A node's ability to expand or collapse is handled with a request to open the node, which is created upon double-clicking it. Then, a flag used in determining whether to show any model children is toggled and the edit part refreshed.

5.5. Navigator

The navigator plug-in adds a tree control outlining the contents of a delta-oriented composite structure diagram or mapping model to the project explorer. That tree control was meant to enable switching to different deltas. However, navigating the current resource is the task of the outline view. Recreating the tree control in the outline view is as good as writing a whole new small editor. Indeed, for composite structure diagrams alone, that tree control comes with its own `ArxTreeEditPart`, `ArxTreeEditPartFactory`, `ArxTreeComponentEditPolicy` and `ArxTreeLayoutEditPolicy`. Double-clicking on a diagram or delta in the tree control creates a request to open it, upon which the editor refreshes the diagram with a new edited delta, effectively switching the view to it. Even though the outline view already fulfills the purpose of the navigator, the navigator plug-in remains as it adds a feature that does not conflict with the rest of the application.

6 Evaluation

To explore the usefulness of the graphical editors, they needs to be evaluated with an elaborate example. This is achieved by modeling the scenarios of the Pick and Place Unit [25].

6.1. Pick and Place Unit

The Pick and Place Unit is an open case study in the field of machine engineering that starts off with a reasonably simple scenario and increases in complexity with every next one. This is useful for gauging how effectively evolutionary variability can be modeled with a given mechanism [12]. In the following, this is done with the graphical editors.

6.1.1. Scenario 0

The first step towards creating an accurate model is setting up the workspace with a new Eclipse project. Assuming Eclipse is up and running, the creation wizard is brought up by pushing the N key while holding the control key. The project of an unspecified type under the general category suffices. Which name the project is given is not important.

Next up is creating the model file. After bringing up the creation wizard again with the same shortcut key, the correct file type is found under the category "Example EMF Model Creation Wizards" as "Arx Model" or "Mapping Model". If it is not there or the entire category is missing, it is most likely that the respective editor plug-in is not installed. The only restriction on the file name is that it needs to have the file extension `arx` or `mapping` to be recognized by the respective graphical editor. The next page asks which model type should be used for the contents, which should always be `Diagram`.

At this point, Eclipse should present the graphical editor with a blank diagram. If it does not, the graphical editor can be opened manually by double-clicking on the newly created file in the project explorer view. The tools from the tool palette on the right are the central method for manipulating the model. Diagram entities and deltas are added by selecting the creation tool of the desired type and clicking on the diagram. The selection tool works with the mouse cursor and doubles as the mechanism for changing location and size by dragging a diagram entity or its border respectively.

Now for actually modeling the first scenario of the Pick and Place Unit. The system to be modeled consists of three main components: a stack, a crane and a ramp. The stack contains work pieces that the crane needs to transport to the ramp. The stack uses two sensors to tell when the separator for supplying a work piece is open and when it is closed as well as another sensor for figuring out whether a work piece is waiting to be picked up. The crane also uses sensors for the position of the crane, that is at the stack or at the ramp, the height of its gripping arm and whether it is currently holding a work piece. The ramp, being a purely mechanical component without input or output, is not considered in the architectural layer. After all, since it does not influence the information flow, it might as well not exist at all from the perspective of a composite structure diagram. The stack notifying the crane of whether a work piece is ready to be picked up constitutes the only interaction

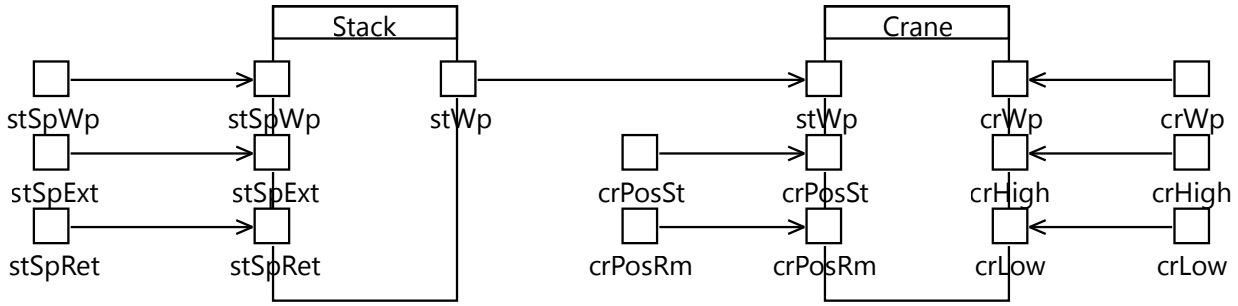

Figure 6.1.: Architecture of Pick and Place Unit scenario 0

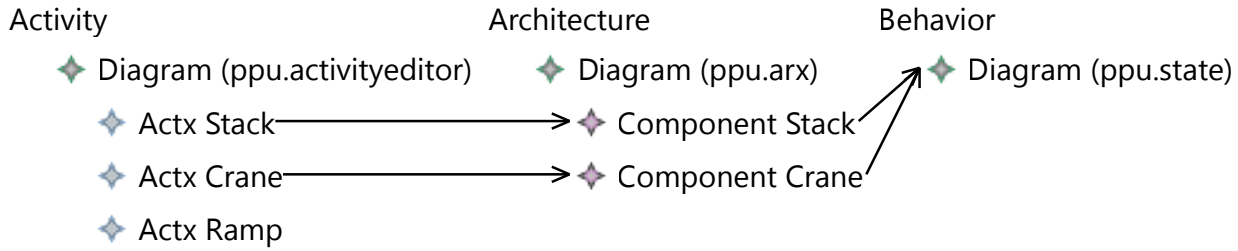

Figure 6.2.: Mapping of Pick and Place Unit scenario 0

between the main components.

Figure 6.1 shows the architecture of this scenario. For the sake of simplicity and to avoid eventually ending up with a huge, cluttered diagram, sensors are not treated as components but as signals from the environment. The names of the entities are not canonical but should still convey the intended meaning.

Figure 6.2 shows the mapping of this scenario. How exactly the activity layer and the behavioral layer are modeled is not relevant to the mapping; only the tasks of the activity layer and the existing models of the behavioral layer need to be known. Each task of the activity layer has a corresponding component of the architectural layer, except for the task of the ramp, for reasons explained above. Finally, all components are linked to the only behavioral model available.

Both of these diagrams serve as the core diagrams for the deltas representing the evolutionary variability in the following scenarios.

6.1.2. Scenario 1

In this scenario, the ramp, formerly just a simple slope, is replaced by a more effective Y-shaped ramp. This does not have an affect on the models as the ramp remains mechanical. Therefore, this scenario does not necessitate the creation of a delta.

6.1.3. Scenario 2

This scenario adds a new type of work pieces to the system. In addition to the normal work pieces, there are now metallic work pieces that are recognized at the stack using an inductive sensor. This information is forwarded to the crane, though as of yet, both work pieces are treated equally and no different course of action is taken by the crane.

To create a delta for this scenario, the delta creation tool needs to be selected and applied by

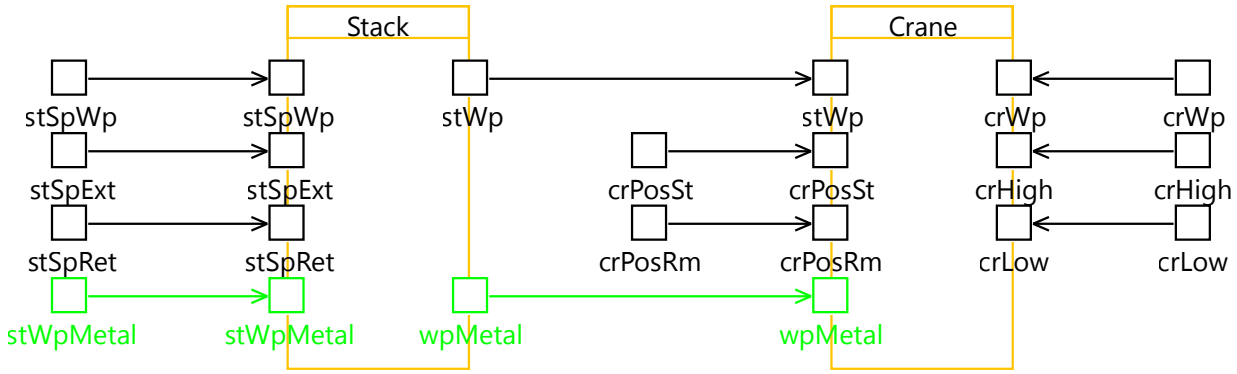

Figure 6.3.: Architecture of Pick and Place Unit scenario 2

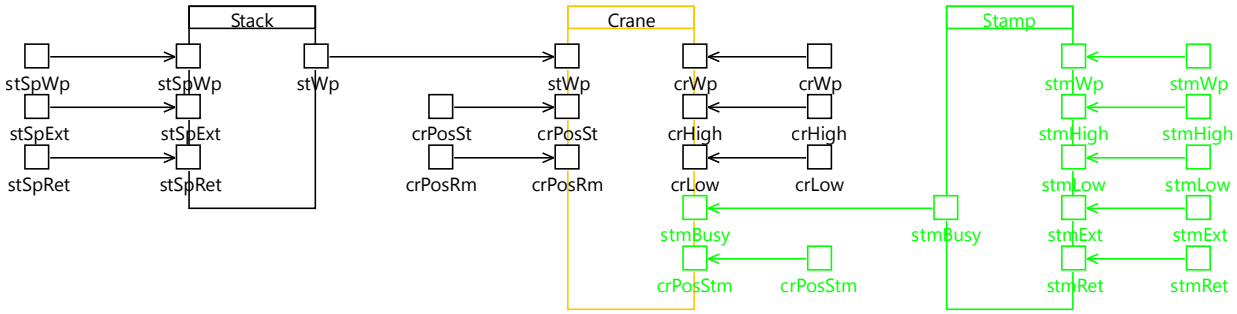

Figure 6.4.: Architecture of Pick and Place Unit scenario 3

clicking on the diagram. The outline view should now list an empty delta in its tree view. Double-clicking on it labels it the delta that is currently being edited. In this mode, manipulating the diagram no longer directly applies the changes but involves delta operations that can be identified by their color. Green is for additions, orange for modifications and red for removals.

Figure 6.3 shows the architectural delta for this scenario. The two components are extended vertically to make room for the additional ports.

The mapping stays the same because no new relevant entities were added, as is true also for most of the coming scenarios.

6.1.4. Scenario 3

The metallic work pieces introduced in the previous scenario are given a purpose with the introduction of an entirely new component: a stamp. The stamp takes any metallic work piece supplied, applies pressure to it and waits for the next one. To this end, it requires a sensor to understand that it was given a work piece, two sensors for positioning the sliding cylinder that moves work pieces to and from the stamping cylinder and two for the stamping cylinder itself exerting the pressure. Instead of taking the metallic work pieces straight to the ramp like it used to, the crane now first brings them to the stamp, waits for it to finish its job and then proceeds taking the work piece to the ramp as per usual. This behavior is not modeled in the architectural layer. Instead, all that matters about this process to the architectural layer is that the crane needs to know when it is positioned correctly for the stamp and for how long the stamp is going to be busy so the work piece can be picked up again afterward.

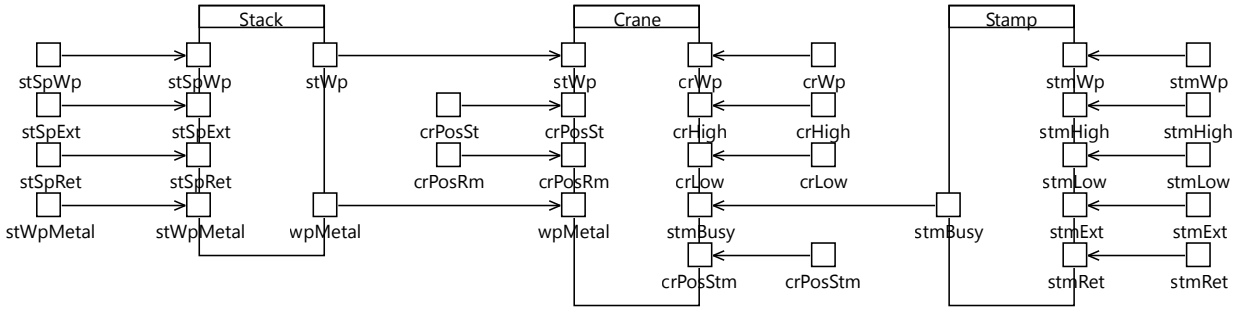

Figure 6.5.: Resolved architecture of Pick and Place Unit scenario 3

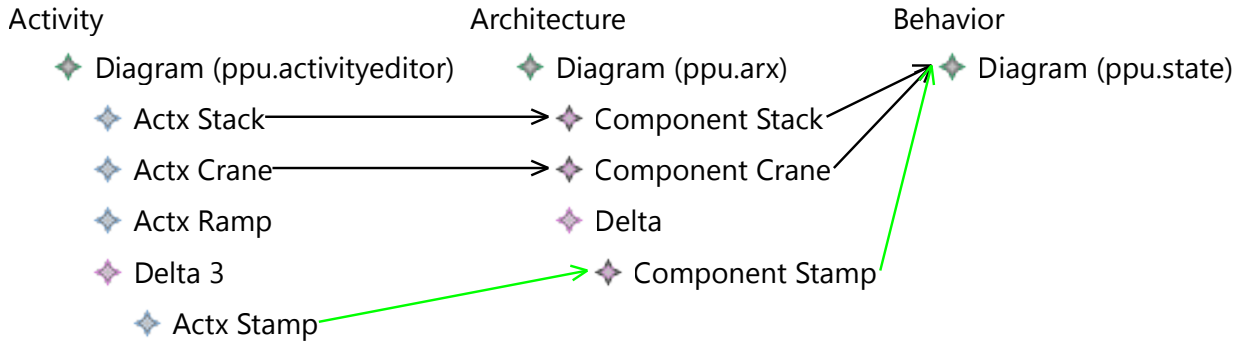

Figure 6.6.: Mapping of Pick and Place Unit scenario 3

Figure 6.4 shows the architectural delta for this scenario. It should be noted that the changes to the diagram from the previous scenario are not visible in this delta. This is because the two deltas are independent of one another. Needless to say, the stamp from this scenario makes little sense without the changes from the previous scenario as it would not ever receive any metallic working pieces to process. In other words, the deltas may be syntactically independent but not semantically. However, it is not the job of the graphical editor to try and understand semantic dependencies. This is why, to make up for the lack of inherent meaning, there exists a mapping in the definition of delta models [16].

Nonetheless, the graphical editor allows something similar to viewing feature configurations. By opening the other delta while this delta is still open, both deltas are shown in a resolved state as can be seen in Figure 6.5. Manipulating this view is disallowed since the answer to the question which delta needs to be updated would be ambiguous. There is also no feature model to back this view, so it is left to the user to decide whether the resolved model is worth being considered meaningful.

Figure 6.6 shows the mapping delta for this scenario. The newly added element is represented on all layers and thus linked in the mapping accordingly.

6.1.5. Scenario 4

Some of the sensors of the crane are optionally replaced by different ones for improved resistance against pollution. Since those sensors send the same signals as the ones they replace, this has no bearing on the architecture layer. Furthermore, no new elements are added, so no change in the mapping occurs.

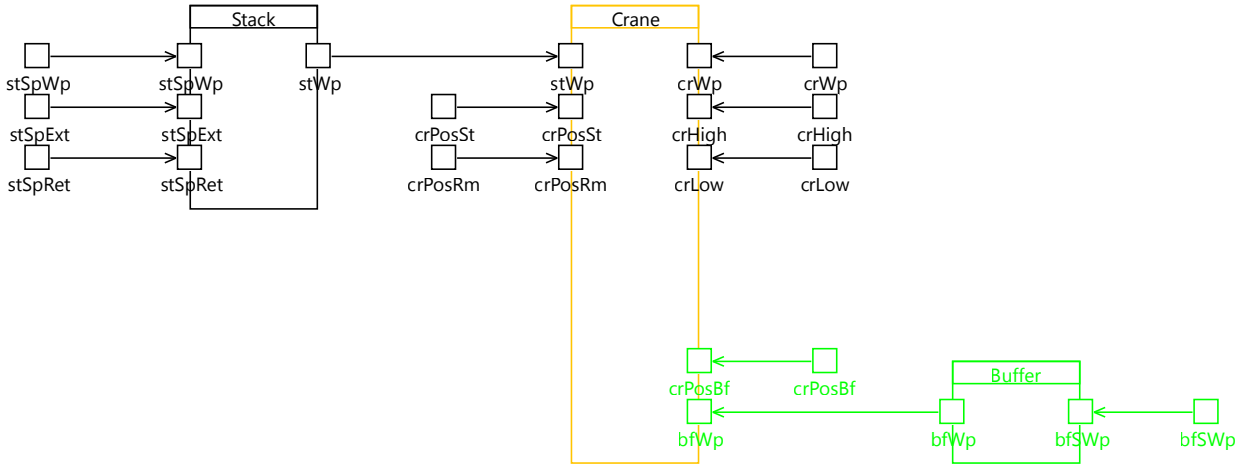

Figure 6.7.: Architecture of Pick and Place Unit scenario 6

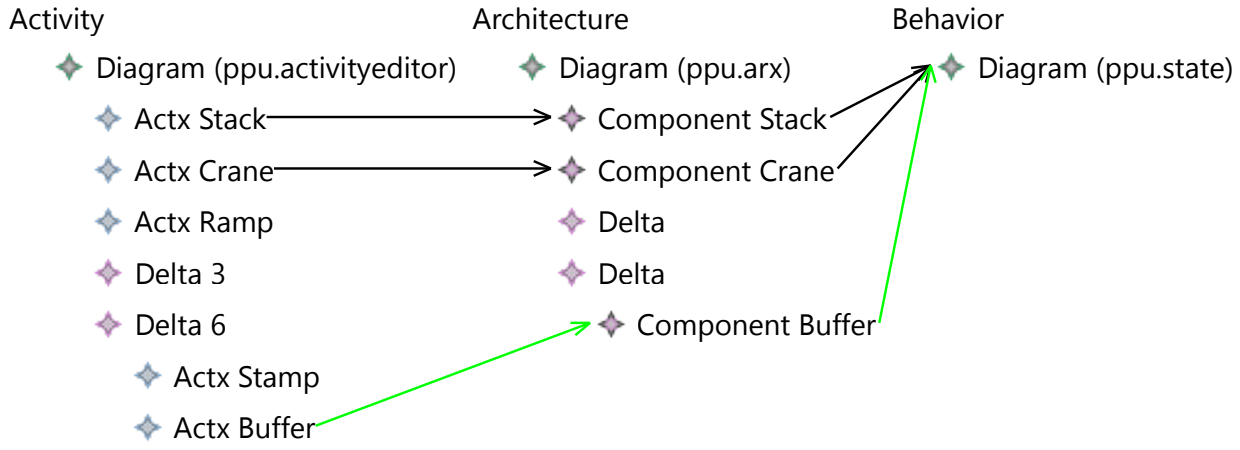

Figure 6.8.: Mapping of Pick and Place Unit scenario 6

6.1.6. Scenario 5

Efficiency of the system is increased by moving work pieces to the ramp in parallel to the stamping process. This is again only a behavioral change.

6.1.7. Scenario 6

A mechanical buffer is added to the stamp, effectively making it capable of holding an additional work piece, which is useful in tandem with the previous scenario's parallelization effort. The crane needs to know whether the buffer is full so it can decide whether it still needs to wait for the buffer to empty before providing another work piece.

Figure 6.7 shows the architectural delta for this scenario. The component denoting the crane is stretched vertically so much to accommodate for the space that the ports from the previous and currently invisible delta use up. Otherwise, resolving these two deltas together would result in some ports being on top of each other. As mentioned already, this is one of the shortcomings of delta-oriented modeling: it is difficult to tell whether something is under the influence of variability at some other point.

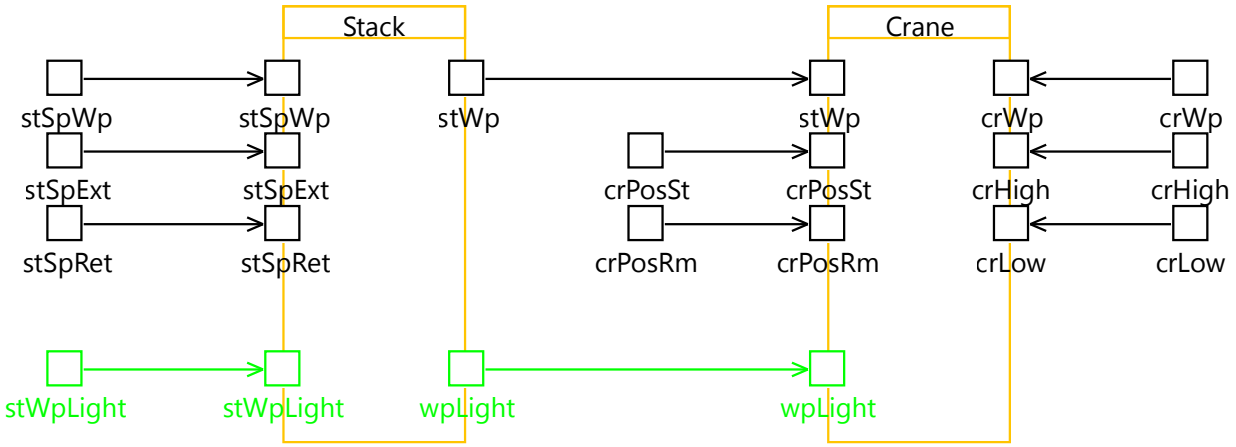

Figure 6.9.: Architecture of Pick and Place Unit scenario 7

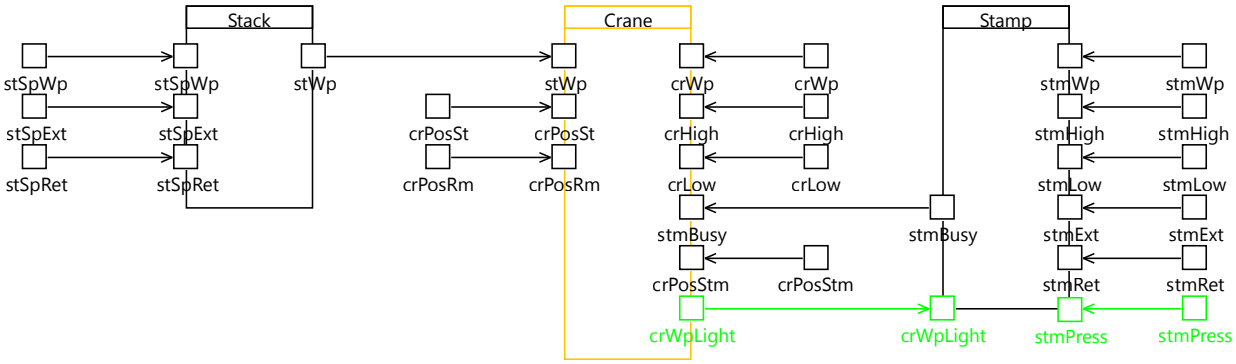

Figure 6.10.: Architecture of Pick and Place Unit scenario 8

Figure 6.8 shows the mapping delta for this scenario. Again, a new element is added and its model representations across the three layers linked.

6.1.8. Scenario 7

Analogously to scenario 3, which added metallic work pieces, this scenario adds white work pieces that are also detected with their own type of sensor, specifically optical sensors.

Figure 6.9 shows the architectural delta for this scenario.

6.1.9. Scenario 8

The white work pieces start being differentiated in the stamp by applying less pressure to them while stamping. This requires a sensor for the stamping pressure and the knowledge of which type the given work piece is.

Figure 6.10 shows the architectural delta for this scenario. This is the first scenario in which the delta depends on another one. In particular, several ports are added to the stamp component from scenario 3. Since only required deltas are shown but a newly created delta does not require any other delta, it is necessary to somehow make the new delta depend on the other. The means for doing so are admittedly somewhat unusual. First, the already existing delta is opened and an addition is made to one of its additions. Then, that operation is moved to the new delta using drag and drop in the outline view. Upon opening the new delta, the dependency is recognized and both deltas are

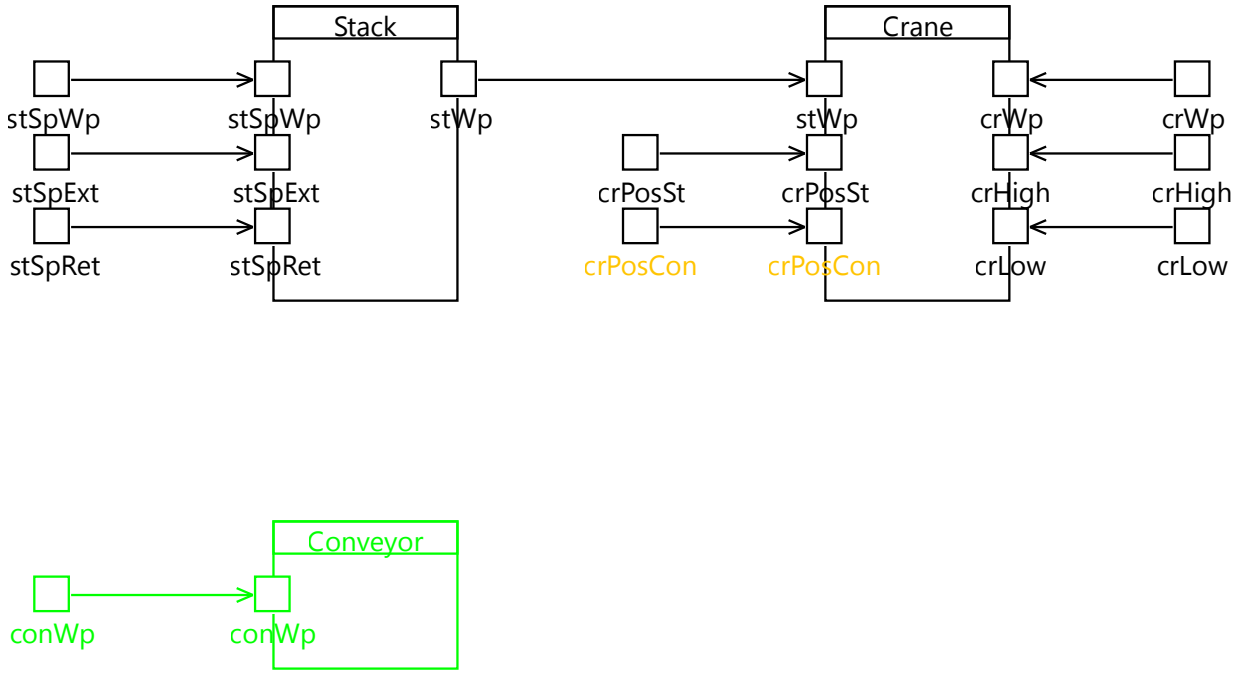

Figure 6.11.: Architecture of Pick and Place Unit scenario 9

visible. From here on, every scenario needing a delta requires this treatment.

6.1.10. Scenario 9

A conveyor belt is put in front of the ramp. The conveyor belt notices when a work piece is put on it and moves it to the end where it drops it into the ramp. Since the conveyor is at the location where the ramp used to be, the crane's sensor for the location of the ramp is relabeled to better reflect its purpose.

Figure 6.11 shows the architectural delta and Figure 6.12 the mapping delta for this scenario.

6.1.11. Scenario 10

The conveyor belt receives two additional ramps. The old ramp at the end of the conveyor belt is filled first, then the ramp in front of it and finally the ramp closest to the crane. To detect where the work pieces are on the conveyor belt and to control pushing the work pieces into the right ramp at the right time, a number of sensors are added.

Figure 6.13 shows the architectural delta and Figure 6.14 the mapping delta for this scenario. Just as with the first scenario, the mechanical ramps are not actually represented with their own components and thus are also not linked in the mapping.

6.1.12. Scenario 11

The conveyor belt now has the task of sorting the work pieces by type into the different ramps. This requires differentiating all possible types at all possible sorting locations, for which two inductive sensors are added to the conveyor belt.

Figure 6.15 shows the delta for this scenario.

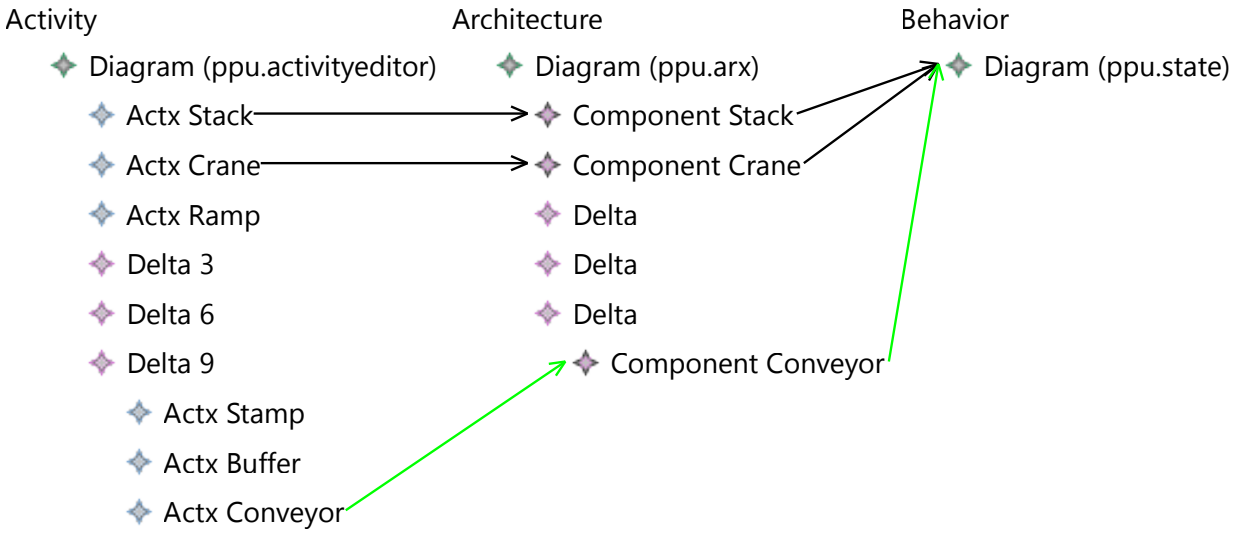

Figure 6.12.: Mapping of Pick and Place Unit scenario 9

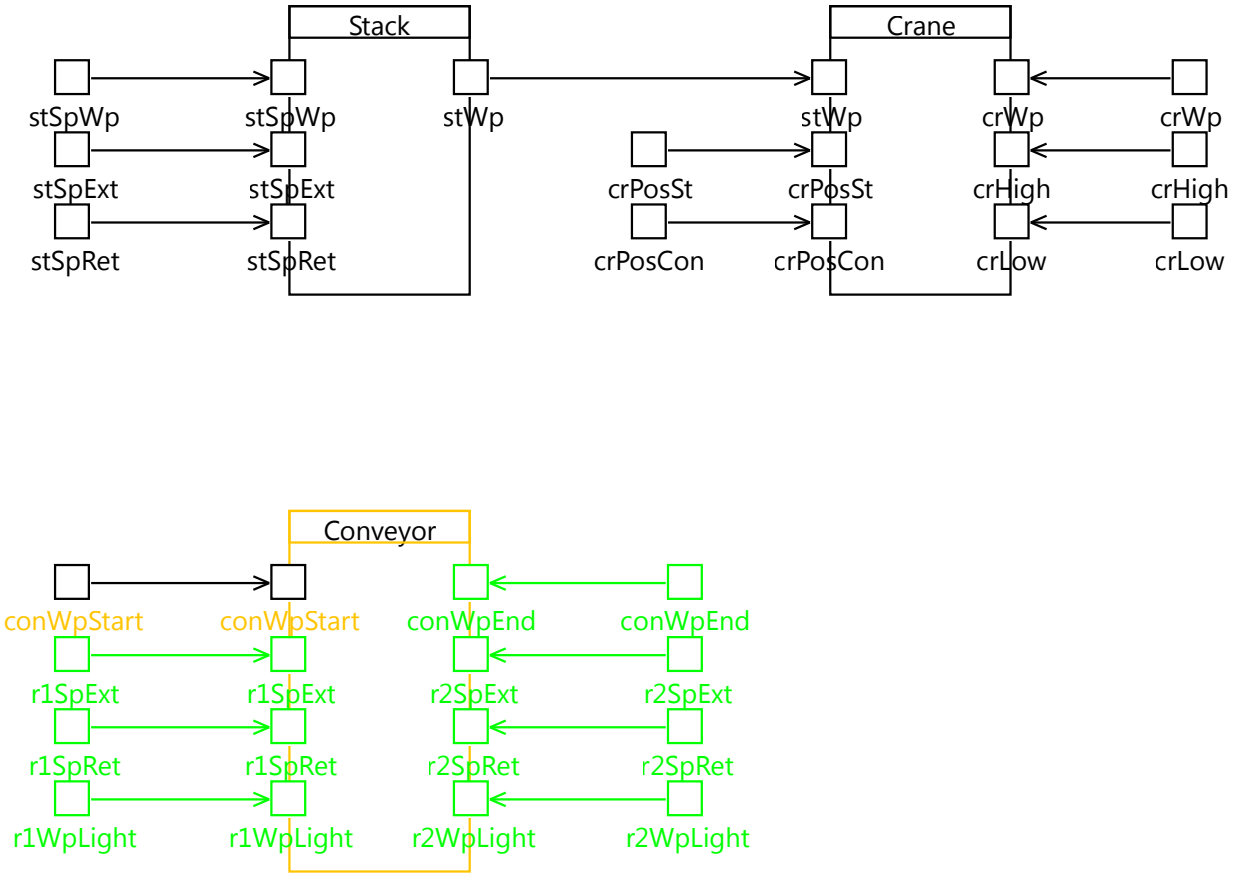

Figure 6.13.: Architecture of Pick and Place Unit scenario 10

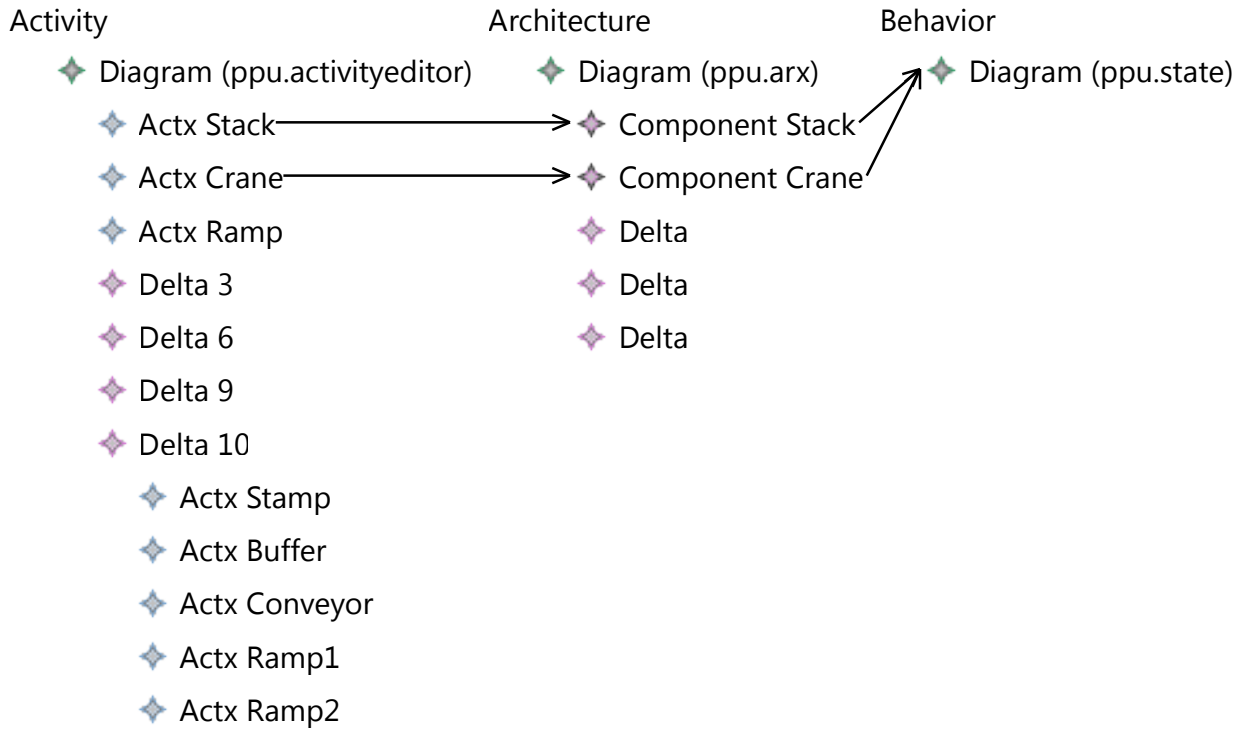

Figure 6.14.: Mapping of Pick and Place Unit scenario 10

6.1.13. Scenario 12

The conveyor belt applies a different sorting algorithm, which is only a behavioral change.

6.1.14. Scenario 13

In this final scenario, the crane's many positioning sensors are replaced by a single potentiometer.

Figure 6.16 shows the delta for this scenario. This is the first scenario in which deletions occur.

Figure 6.17 and Figure 6.18 show the final resolved models with all deltas enabled. Though this use case is simple enough, it still is apparent that the delta-oriented variability approach greatly reduces the complexity in each evolutionary step by masking all the currently unnecessary information.

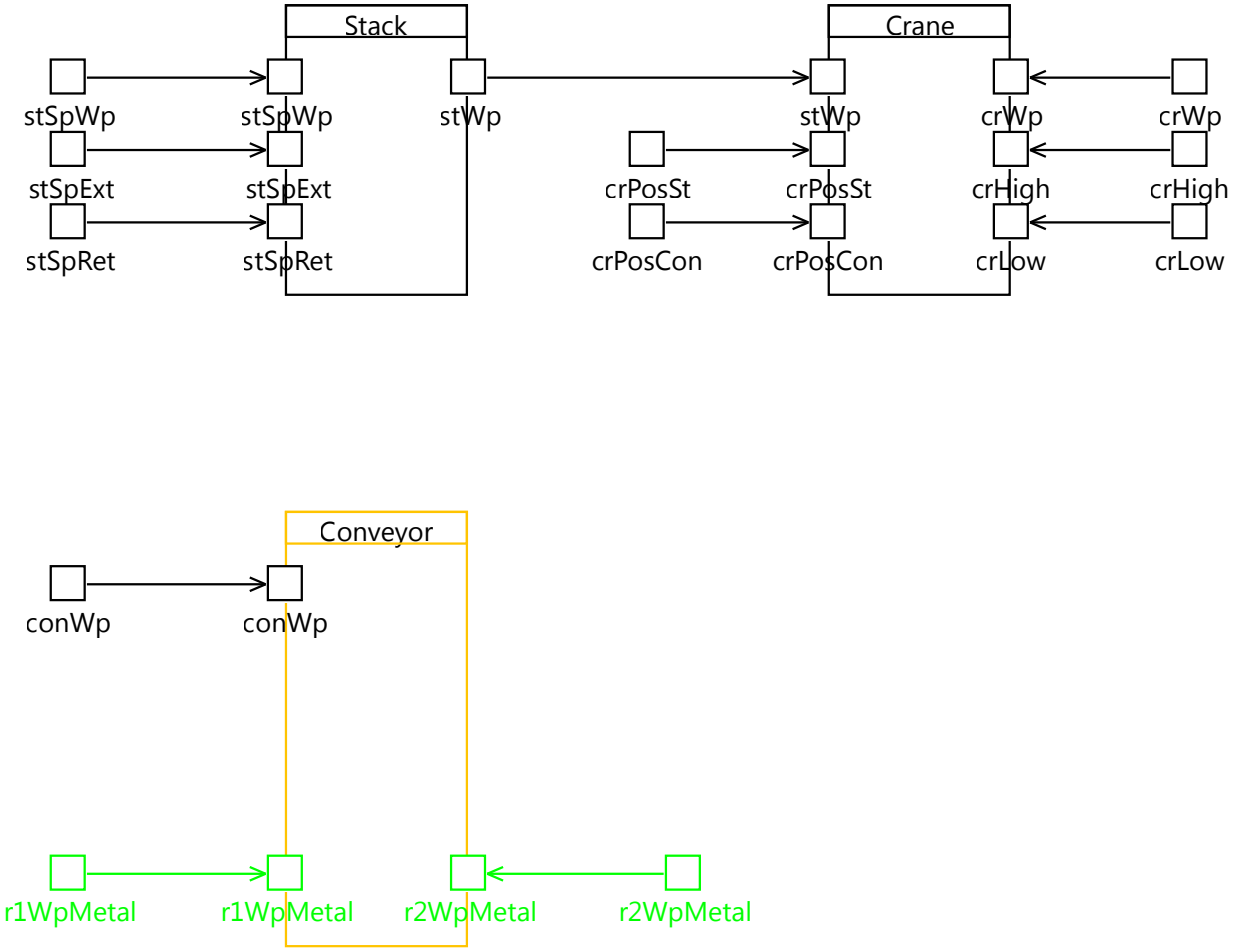

Figure 6.15.: Architecture of Pick and Place Unit scenario 11

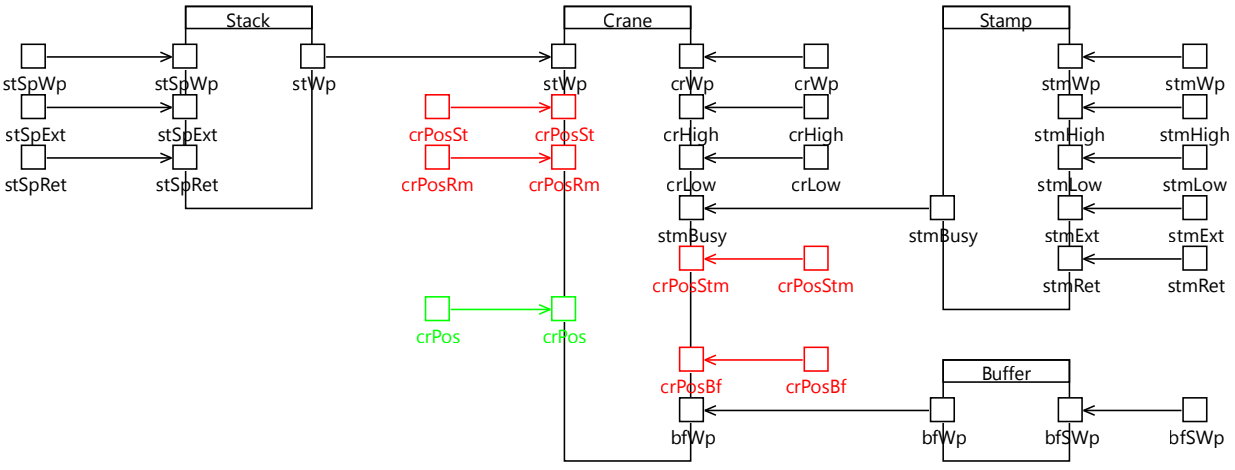

Figure 6.16.: Architecture of Pick and Place Unit scenario 13

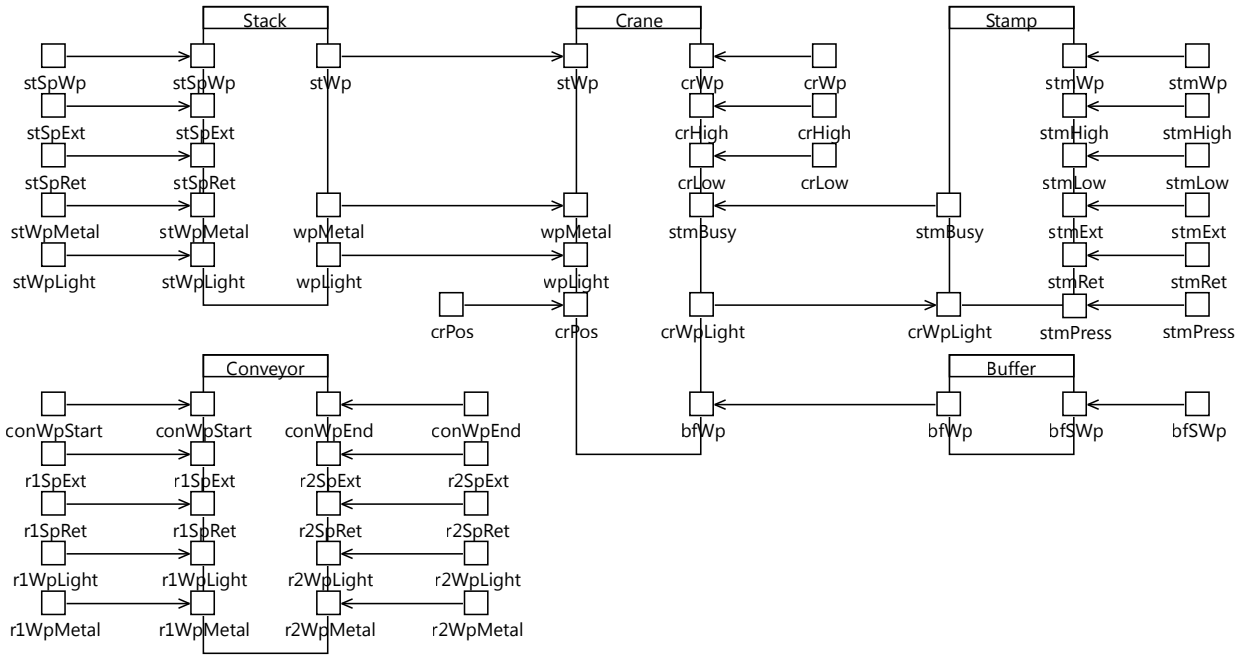

Figure 6.17.: Resolved architecture of Pick and Place Unit scenario 13

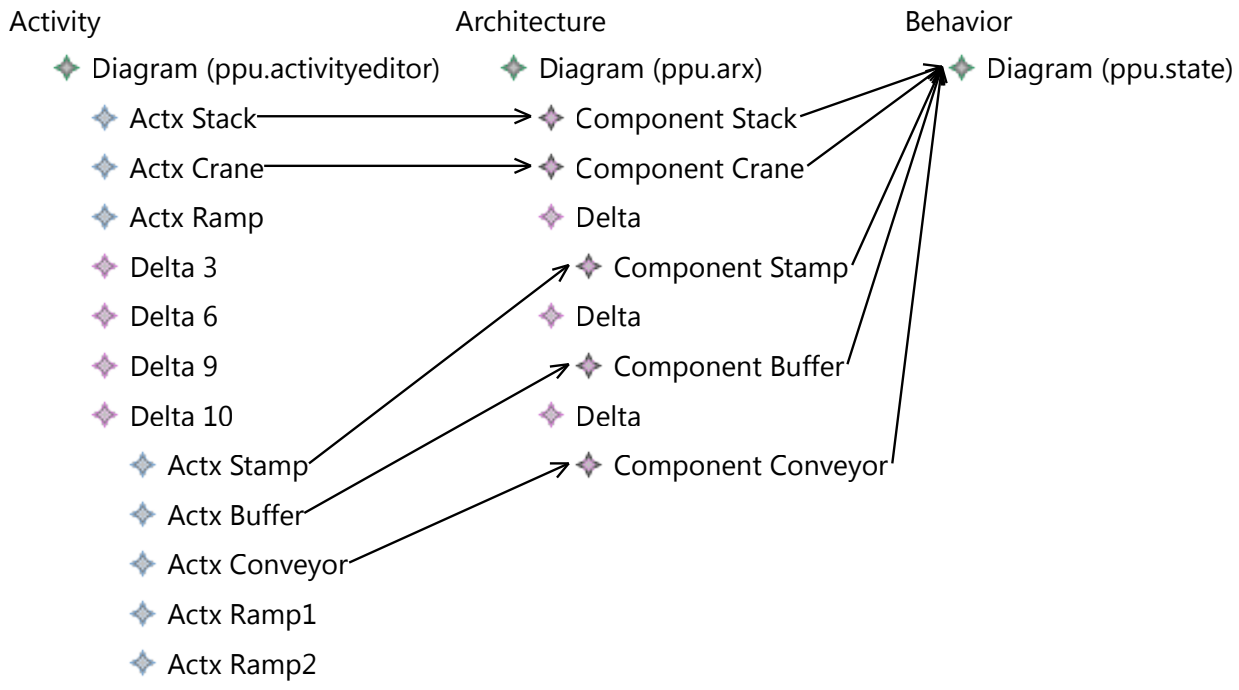

Figure 6.18.: Resolved mapping of Pick and Place Unit scenario 13

7 Conclusion

This bachelor thesis has shown how to graphically realize delta-oriented modeling on three layers of abstraction. This involved the development of a graphical editor for delta-oriented composite structure diagrams and another for mapping models.

The first step in the development was the analysis. During the analysis, a number of approaches for modeling variability in software systems were examined. Of particular importance to the development of the editor was the visual representation of the model in the various modeling approaches. These views were compared by various aspects such how intuitive they are and how well they separate the model from the variability mechanism. Unsurprisingly, it turned out that every approach has different advantages and disadvantages.

After the analysis, fundamental design decisions about the graphical editors were made. On the technical level, most of these were adopted from previous related work [23, 15]. However, the design of the views of the models required special attention. The knowledge gained from the analysis proved useful here. The idea was to keep the views as simple as possible. Through masking unnecessary information while marking the currently relevant variability with easily recognizable colors, intuitive views of delta-oriented composite structure diagrams and mapping models were developed.

Subsequently, the implementation itself was achieved with the help of EMF and GEF. With EMF responsible for the model, Draw2d for the view and GEF for the controller, a piece of software with proper separation of concerns and thus maintainability was created. The numerous issues encountered during this process were analyzed and solved or at least countered sensibly in the case of more fundamental issues such as missing support for feature refinement by EMF.

Finally, for the evaluation, the newly created application was tested by modeling the scenarios of the Pick and Place Unit. Capturing the evolutionary variability of each scenario was as simple and intuitive as hoped. All delta operations work as intended and are visualized correctly. The graphical editor for composite structure diagrams also supports the creation of hierarchically structured components, which was not even deemed absolutely necessary for modeling the architectural layer. Likewise, the mapping editor proved being a handy tool for creating and editing mapping models due to its compact tree view and easy point and click linking. Therefore it is safe to say that the goal of this bachelor thesis is met.

In the future, the previous two and the new two graphical editors will be integrated into a bigger one working with multiple layers. Before or at least by the time that happens, the models could most likely still be made more sound and maintainable assuming advanced knowledge of EMF. Additionally, in order to support the DSL combining the three model layers [16], export wizards for models of the architectural layer and for mapping models need to be implemented.

Bibliography

[1] A. Antonenko. “Entwicklung und Evaluierung eines grafischen Modellierungsansatzes für delta-orientierte Aktivitätsdiagramme”. Bachelor thesis. Technische Universität Braunschweig, 08/2015.

[2] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 10/2013.

[3] O. Avila-García, A. E. García, and E. V. S. Rebull. “Using Software Product Lines to Manage Model Families in Model-Driven Engineering”. In: *ACM SAC 2008*. 03/2007, pp. 1006–1011.

[4] K. Czarnecki and M. Antkiewicz. “Mapping Features to Models: A Template Approach Based on Superimposed Variants”. In: *GPCE 2005*. 09/2005, pp. 422–437.

[5] R. B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg. “Model-Driven Development Using UML 2.0: Promises and Pitfalls”. In: *Computer* 39.2 (02/2006), pp. 59–66.

[6] H. Gomaa. “Designing Software Product Lines with UML 2.0: From Use Cases to Pattern-Based Software Architectures”. In: *SPLC 2006*. 08/2006, pp. 218–218.

[7] J. Gosling and H. McGilton. *The Java Language Environment*. 05/1996.

[8] R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 03/2009.

[9] M. Harsu. *A Survey on Domain Engineering*. Tampere University of Technology, 12/2002.

[10] O. Haugen, B. Moller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen. “Adding Standardized Variability to Domain Specific Languages”. In: *SPLC 2008*. 09/2008, pp. 139–148.

[11] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. Software Engineering Institute, Carnegie Mellon University, 11/1990.

[12] M. Kowal, C. Legat, D. Lorefice, C. Prehofer, I. Schaefer, and B. Vogel-Heuser. “Delta Modeling for Variant-rich and Evolving Manufacturing Systems”. In: *MoSEMInA 2014*. 05/2014, pp. 32–41.

[13] R. Lachmann. “Konzeption und Evaluation eines delta-orientierten modellbasierten Integrationstestverfahrens für Softwareproduktlinien”. Master thesis. Technische Universität Braunschweig, 04/2012.

[14] J. Lee, N.-L. Xue, and T.-L. Kuei. “A Note on State Modeling Through Inheritance”. In: *SIGSOFT Software Engineering Notes* 23.1 (01/1998), pp. 104–110.

[15] S. Lity. “Konzeption und Evaluation eines delta-orientierten modellbasierten Testverfahrens für Softwareproduktlinien”. Master thesis. Technische Universität Braunschweig, 12/2011.

[16] D. Lorefice. “Delta-orientierte Modellierung und Konsistenzüberprüfung über mehrere Sichten”. Master thesis. Technische Universität Braunschweig, 05/2014.

[17] C. Meyer. “Graphische Darstellung von Deltas in UML-Statechart Diagrammen”. Bachelor thesis. Technische Universität Braunschweig, 09/2014.

[18] W. Pree, M. Fontoura, and B. Rumpe. “Product Line Annotations with UML-F”. In: *SPLC 2002*. 08/2002, pp. 188–197.

[19] I. Schaefer. “Variability Modelling for Model-Driven Development of Software Product Lines”. In: *VaMoS 2010*. 01/2010, pp. 85–92.

[20] A. Schnieders and F. Puhlmann. “Activity Diagram Inheritance”. In: *BIS 2005*. 04/2005, pp. 435–448.

[21] B. Selic. “The Pragmatics of Model-Driven Development”. In: *IEEE Software* 20.5 (09/2003), pp. 19–25.

[22] A. J. H. Simons, M. P. Stannett, K. E. Bogdanov, and W. M. L. Holcombe. “Plug and Play Safely: Rules for Behavioural Compatibility”. In: *IASTED SEA 2012*. 11/2002, pp. 263–268.

[23] C. Singer. “Delta-Modellierung für Aktivitätsdiagramme”. Project work. Technische Universität Braunschweig, 06/2013.

[24] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. “A Classification and Survey of Analysis Strategies for Software Product Lines”. In: *ACM Computing Surveys* 47.1 (06/2014), 6:1–6:45.

[25] B. Vogel-Heuser, C. Legat, J. Folmer, and S. Feldmann. *Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the Pick and Place Unit*. Tech. rep. TUM-AIS-TR-01-14-02. Institute of Automation and Information Systems, Technische Universität München, 2014.

Eidesstattliche Erklärung

Ich versichere, dass ich die beiliegende Bachelorarbeit ohne Hilfe Dritter und ohne Benutzung anderer, als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher Form noch keiner Prüfungsbehörde vorgelegen.
Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Ort, Datum Unterschrift

Technische Universität Carolo-Wilhelmina in Braunschweig (Germany)
Institute of Software Engineering and Automotive Informatics

Mühlenpfordtstr. 23
D-38106 Braunschweig